# Contents

**Preface** ..........................................................................................................................11

- Documentation Conventions ......................................................................................12
- Related Publications .....................................................................................................13
- Customer Support .........................................................................................................13
- Information You Should Have ......................................................................................14
- User Feedback ................................................................................................................15
- Information Builders Consulting and Training .............................................................15

**1. Introducing Functions** .......................................................................................17

- Using Functions .............................................................................................................18
- Types of Functions .........................................................................................................19
  - Character Functions ..................................................................................................20
  - Variable Length Character Functions ....................................................................22
  - Character Functions for DBCS Code Pages ..............................................................23
  - Maintain-specific Character Functions ....................................................................23
  - Data Source and Decoding Functions .....................................................................25
  - Date Functions ..........................................................................................................25
  - Date-Time Functions .................................................................................................28
  - Maintain-specific Date and Time Functions .............................................................29
  - Format Conversion Functions ..................................................................................30
  - Numeric Functions .....................................................................................................31
  - System Functions .......................................................................................................33
  - Character Chart for ASCII and EBCDIC ..................................................................34

**2. Accessing and Calling a Function** ....................................................................43

- Calling a Function .........................................................................................................44
- Supplying an Argument in a Function ..........................................................................46
  - Argument Types .......................................................................................................46
  - Argument Formats .....................................................................................................47
  - Argument Length .......................................................................................................48
3. Character Functions.......................................................................................67

Character Function Notes..................................................................................68
ARGLEN: Measuring the Length of a String.......................................................68
ASIS: Distinguishing Between Space and Zero..................................................69
BITSON: Determining If a Bit Is On or Off.........................................................71
BITVAL: Evaluating a Bit String as an Integer...................................................72
BYPVAL: Translating a Character to Decimal.....................................................73
CHKFMT: Checking the Format of a String.........................................................75
CTRAN: Translating One Character to Another...................................................78
CTRFLD: Centering a Character String...............................................................84
EDIT: Extracting or Adding Characters...............................................................85
GETTOK: Extracting a Substring (Token).............................................................87
LCWORD: Converting a String to Mixed-Case....................................................89
LCWORD2: Converting a String to Mixed-Case..................................................91
LCWORD3: Converting a String to Mixed-Case..................................................92
LJUST: Left-Justifying a String...........................................................................93
LOCASE: Converting Text to Lowercase.............................................................95
OVRLAY: Overlaying a Character String..........................................................96
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LJUST: Left-Justifying a Character String</td>
<td>155</td>
</tr>
<tr>
<td>LOWER: Converting a Character String to Lowercase</td>
<td>156</td>
</tr>
<tr>
<td>MASK: Extracting or Adding Characters</td>
<td>156</td>
</tr>
<tr>
<td>MNTGETTOK: Extracting Tokens From a String Function</td>
<td>158</td>
</tr>
<tr>
<td>NLSCHR: Converting Characters From the Native English Code Page</td>
<td>162</td>
</tr>
<tr>
<td>OVRLAY: Overlaying a Character String</td>
<td>162</td>
</tr>
<tr>
<td>POSIT: Finding the Beginning of a Substring</td>
<td>164</td>
</tr>
<tr>
<td>RJUST: Right-Justifying a Character String</td>
<td>165</td>
</tr>
<tr>
<td>SELECTS: Decoding a Value From a Stack</td>
<td>166</td>
</tr>
<tr>
<td>STRAN: Substituting One Substring for Another</td>
<td>167</td>
</tr>
<tr>
<td>STRCMP: Comparing Character Strings</td>
<td>169</td>
</tr>
<tr>
<td>STRICMP: Comparing Character Strings and Ignoring Case</td>
<td>171</td>
</tr>
<tr>
<td>STRNCMP: Comparing Character Substrings</td>
<td>171</td>
</tr>
<tr>
<td>STRTOKEN: Extracting a Substring Based on Delimiters</td>
<td>172</td>
</tr>
<tr>
<td>SUBSTR: Extracting a Substring</td>
<td>173</td>
</tr>
<tr>
<td>TRIM: Removing Trailing Occurrences</td>
<td>175</td>
</tr>
<tr>
<td>TRIMLEN: Determining the Length of a String Excluding Trailing Spaces</td>
<td>175</td>
</tr>
<tr>
<td>UPCASE: Converting Text to Uppercase</td>
<td>176</td>
</tr>
</tbody>
</table>

7. Data Source and Decoding Functions

- DB_LOOKUP: Retrieving Data Source Values...178
- DECODE: Decoding Values..................181
- FIND: Verifying the Existence of a Value in a Data Source...185
- LAST: Retrieving the Preceding Value........189
- LOOKUP: Retrieving a Value From a Cross-referenced Data Source...191
  Using the Extended LOOKUP Function........196

8. Date Functions

- Overview of Date Functions...200
- Using Standard Date Functions........201
  Specifying Work Days.....................202
  Enabling Leading Zeros For Date and Time Functions in Dialogue Manager...204
- DATEADD: Adding or Subtracting a Date Unit to or From a Date...206
- DATECVT: Converting the Format of a Date...209
- DATEDIF: Finding the Difference Between Two Dates........211
9. Date-Time Functions .......................................................... 263

Using Date-Time Functions ...................................................... 264
  Date-Time Parameters .......................................................... 264
  Supplying Arguments for Date-Time Functions ...................... 267
  Using Date-Time Formats ..................................................... 269
  Assigning Date-Time Values .................................................. 271
HADD: Incrementing a Date-Time Value ...................................... 274
HCNVRT: Converting a Date-Time Value to Alphanumeric Format ............................................................................ 276
HDATE: Converting the Date Portion of a Date-Time Value to a Date Format .......................................................... 278
HDIFF: Finding the Number of Units Between Two Date-Time Values ........................................................................... 279
HDTTM: Converting a Date Value to a Date-Time Value ................ 281
HEXTR: Extracting Components of a Date-Time Value and Setting Remaining Components to Zero ........................................... 282
10. Maintain-specific Date and Time Functions ...............................................305

Maintain-specific Standard Date and Time Functions ...........................................306

HHMMSS: Retrieving the Current Time (Maintain). ..................................................306
Initial_HHMMSS: Returning the Time the Application Was Started .......................307
Initial_TODAY: Returning the Date the Application Was Started ..........................307
TODAY: Retrieving the Current Date (Maintain) ....................................................308
TODAY2: Returning the Current Date .................................................................309
ADD: Adding Days to a Date ................................................................................309
DAY: Extracting the Day of the Month From a Date ..............................................310
JULIAN: Determining How Many Days Have Elapsed in the Year .......................311
MONTH: Extracting the Month From a Date .........................................................312
QUARTER: Determining the Quarter ..................................................................312
SETMDY: Setting the Value to a Date .................................................................313
SUB: Subtracting a Value From a Date ...............................................................314
WEEKDAY: Determining the Day of the Week for a Date ....................................315
YEAR: Extracting the Year From a Date ..............................................................316

11. Format Conversion Functions .....................................................................317

ATODBL: Converting an Alphanumeric String to Double-Precision Format ..........318
EDIT: Converting the Format of a Field ...............................................................321
FPRINT: Converting Fields to Alphanumeric Format .............................................322
FTOA: Converting a Number to Alphanumeric Format ........................................328
12. Numeric Functions

ABS: Calculating Absolute Value.................................................................344
ASIS: Distinguishing Between a Blank and a Zero........................................345
BAR: Producing a Bar Chart........................................................................345
CHKPCK: Validating a Packed Field............................................................347
DMOD, FMOD, and IMOD: Calculating the Remainder From a Division........350
EXP: Raising e to the Nth Power.................................................................351
EXPN: Evaluating a Number in Scientific Notation........................................352
FMLINFO: Returning FOR Values...............................................................353
FMLLIST: Returning an FML Tag List.........................................................355
FMLFOR: Retrieving FML Tag Values.........................................................357
FMLCAP: Retrieving FML Hierarchy Captions.............................................358
INT: Finding the Greatest Integer...............................................................359
LOG: Calculating the Natural Logarithm.......................................................360
MAX and MIN: Finding the Maximum or Minimum Value...........................361
MIRR: Calculating the Modified Internal Return Rate.................................362
NORMSDST: Calculating Standard Cumulative Normal Distribution..........365
NORMSINV: Calculating Inverse Cumulative Normal Distribution..............368
PRDNOR and PRDUNI: Generating Reproducible Random Numbers..........370
RDNORM and RDUNIF: Generating Random Numbers.............................372
SQRT: Calculating the Square Root............................................................373
XIRR: Calculating the Modified Internal Return Rate (Periodic or Non-Periodic)........................................................................374

13. System Functions

CLSDDREC: Closing All Files Opened by the PUTDDREC Function............380
FEXERR: Retrieving an Error Message.......................................................380
Contents

FINDMEM: Finding a Member of a Partitioned Data Set....................................................381
GETPDS: Determining If a Member of a Partitioned Data Set Exists............................383
GETUSER: Retrieving a User ID.......................................................................................388
MVSDYNNAM: Passing a DYNAM Command to the Command Processor......................389
PUTDDREC: Writing a Character String as a Record in a Sequential File..........................391
SLEEP: Suspending Execution for a Given Number of Seconds.........................................394
SYSVAR: Retrieving the Value of a z/OS System Variable.............................................396

A. Creating a Subroutine.................................................................................................399
  Writing a Subroutine......................................................................................................400
    Naming a Subroutine.................................................................................................401
    Creating Arguments.................................................................................................401
    Language Considerations.........................................................................................402
    Programming a Subroutine......................................................................................405
  Compiling and Storing a Subroutine............................................................................411
    Compiling and Storing a Subroutine on z/OS..........................................................411
  Testing the Subroutine.................................................................................................411
  Using a Custom Subroutine: The MTHNAM Subroutine..............................................412
    Writing the MTHNAM Subroutine...........................................................................412
    Calling the MTHNAM Subroutine From a Request..................................................426
  Subroutines Written in REXX......................................................................................427
    Formats and REXX Subroutines..............................................................................433

Reader Comments..........................................................................................................451
Preface

This documentation describes how to use functions to perform certain calculations and manipulations. It is intended for application developers. This manual is part of the FOCUS documentation set.

The documentation set consists of the following components:

- The Creating Reports manual describes FOCUS Reporting environments and features.
- The Describing Data manual explains how to create the metadata for the data sources that your FOCUS procedures will access.
- The Developing Applications manual describes FOCUS Application Development tools and environments.
- The Maintaining Databases manual describes FOCUS data management facilities and environments.
- The Using Functions manual describes internal functions and user-written subroutines.
- The Overview and Operating Environments manual contains an introduction to FOCUS and FOCUS tools and describes how to use FOCUS in the z/OS environment.

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><strong>1</strong></td>
<td>Introducing Functions</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>Accessing and Calling a Function</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>Character Functions</td>
</tr>
</tbody>
</table>

Introduces functions and explains the different types of available functions.

Describes the considerations for supplying arguments in a function, and explains how to use a function in a command and access functions stored externally.

Describes character functions that manipulate alphanumeric fields and character strings.
<table>
<thead>
<tr>
<th>Chapter/Appendix</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4</strong></td>
<td>Variable Length Character Functions</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>Character Functions for DBCS Code Pages</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>Maintain-specific Character Functions</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>Data Source and Decoding Functions</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>Date Functions</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>Date-Time Functions</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>Maintain-specific Date and Time Functions</td>
</tr>
<tr>
<td><strong>11</strong></td>
<td>Format Conversion Functions</td>
</tr>
<tr>
<td><strong>12</strong></td>
<td>Numeric Functions</td>
</tr>
<tr>
<td><strong>13</strong></td>
<td>System Functions</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>Creating a Subroutine</td>
</tr>
</tbody>
</table>

**Documentation Conventions**

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

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

## Related Publications

You can also contact the Publications Order Department at (800) 969-4636.

## Customer Support

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

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

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

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

**Information You Should Have**

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

- Your six-digit site code (xxxx.xx).
- The stored procedure (preferably with line numbers).
- The Master File and Access File.
- Run sheet (beginning at login, including call to FOCUS), containing the following information:
  - ? RELEASE
  - ? FDT
  - ? LET
  - ? LOAD
  - ? COMBINE
  - ? JOIN
  - ? DEFINE
Preface

- ? STAT
- ? SET/? SET GRAPH
- ? TSO DDNAME

- The exact nature of the problem:
  - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
  - The error message and return code, if applicable.
  - Is this related to any other problem?

- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?

- What release of the operating system are you using? Has it, FOCUS, your security system, or an interface system 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. Please use the Reader Comments form at the end of this document to communicate your feedback to us or to suggest changes that will support improvements to our documentation. You can also contact us through our website http://documentation.informationbuilders.com/connections.asp.

Thank you, in advance, for your comments.

**Information Builders Consulting and Training**

Interested in training? Information Builders Education Department offers a wide variety of training courses for this and other Information Builders products.
For information on course descriptions, locations, and dates, or to register for classes, visit our website (http://education.informationbuilders.com) or call (800) 969-INFO to speak to an Education Representative.
Introducing Functions

The following topics offer an introduction to functions and explain the different types of functions available.

Topics:
- Using Functions
- Types of Functions
- Character Chart for ASCII and EBCDIC
Using Functions

Functions operate on one or more arguments and return a single value. The returned value can be stored in a field, assigned to a Dialogue Manager variable, used in a calculation or other processing, or used in a selection or validation test. Functions provide a convenient way to perform certain calculations and manipulations.

There are three types of functions:

- **Internal functions.** Built into the FOCUS language, requiring no extra work to access or use. The following reporting and Maintain functions are internal functions. You can not replace any of these internal functions with your own functions of the same name. All other functions are external.
  - ABS
  - ASIS
  - DMY, MDY, and YMD
  - DECODE
  - EDIT
  - FIND
  - LAST
  - LOG
  - LOOKUP
  - MAX and MIN
  - SQRT
  - All Maintain-specific functions

- **External functions.** Stored in an external library that must be accessed. When invoking these functions, an argument specifying the output field or format of the result is required. External functions are distributed with FOCUS. You can replace these functions with your own functions of the same name. However, in this case, you must set USERFNS=LOCAL.

- **Subroutines.** Written by the user and stored externally. For details, see Creating a Subroutine on page 399.

For information on how to use an internal or external function, see Accessing and Calling a Function on page 43.
Types of Functions

In this section:
Character Functions
Variable Length Character Functions
Character Functions for DBCS Code Pages
Maintain-specific Character Functions
Data Source and Decoding Functions
Date Functions
Date-Time Functions
Maintain-specific Date and Time Functions
Format Conversion Functions
Numeric Functions
System Functions

You can access any of the following types of functions:

- **Character functions.** Manipulate alphanumeric fields or character strings. For details, see Character Functions on page 20.
- **Variable length character functions.** Manipulate AnV fields or character strings. For details, see Variable Length Character Functions on page 22.
- **Character functions for DBCS code pages.** Manipulate alphanumeric fields or character strings on DBCS code pages. For details, see Character Functions for DBCS Code Pages on page 23.
- **Maintain-specific character functions.** Manipulate alphanumeric fields or character strings. These functions are available only in Maintain. For details, see Maintain-specific Character Functions on page 23.
- **Data source and decoding functions.** Search for or retrieve data source records or values, and assign values. For details, see Data Source and Decoding Functions on page 25.
- **Date functions.** Manipulate dates. For details, see Date Functions on page 25.
- **Date-time functions.** Manipulate date-time values. For details, see Date-Time Functions on page 28.
Types of Functions

- **Maintain-specific date and time functions.** Manipulate dates and times. These functions are available only in Maintain. For details, see *Maintain-specific Date and Time Functions* on page 29.

- **Format conversion functions.** Convert fields from one format to another. For details, see *Format Conversion Functions* on page 30.

- **Numeric functions.** Perform calculations on numeric constants and fields. For details, see *Numeric Functions* on page 31.

- **System functions.** Call the operating system to obtain information about the operating environment or to use a system service. For details, see *System Functions* on page 33.

**Character Functions**

The following functions manipulate alphanumeric fields or character strings. For details, see *Character Functions* on page 67.

**ARGLEN**

Measures the length of a character string within a field, excluding trailing blanks.

**ASIS**

Distinguishes between a blank and a zero in Dialogue Manager.

**BITSON**

Evaluates an individual bit within a character string to determine whether it is on or off.

**BITVAL**

Evaluates a string of bits within a character string and returns its value.

**BYTVAL**

Translates a character to its corresponding ASCII or EBCDIC decimal value.

**CHKFMT**

Checks a character string for incorrect characters or character types.

**CTRAN**

Translates a character within a character string to another character based on its decimal value.

**CTRFLD**

Centers a character string within a field.

**EDIT**

Extracts characters from or adds characters to a character string.
**GETTOK**
Divides a character string into substrings, called tokens, where a specific character, called a delimiter, occurs in the string.

**LCWORD**
Converts the letters in a character string to mixed case.

**LCWORD2**
Converts the letters in a character string to mixed case.

**LCWORD3**
Converts the letters in a character string to mixed case.

**LJUST**
Left-justifies a character string within a field.

**LOCASE**
Converts alphanumeric text to lowercase.

**OVRLAY**
Overlays a base character string with a substring.

**PARAG**
Divides a line of text into smaller lines by marking them with a delimiter.

**POSIT**
 Finds the starting position of a substring within a larger string.

**REVERSE**
Reverses the characters in a character string.

**RJUST**
Right-justifies a character string.

**SOUNDEX**
Searches for a character string phonetically without regard to spelling.

**SPELLNM**
Takes an alphanumeric string or a numeric value with two decimal places and spells it out with dollars and cents. This function is available only for WebFOCUS.

**SQUEEZ**
Reduces multiple contiguous spaces within a character string to a single space.
**Types of Functions**

**STRIP**
Removes all occurrences of a specific character from a string.

**STRREP**
Replaces all occurrences of a specific character string.

**SUBSTR**
Extracts a substring based on where it begins and its length in the parent string.

**TRIM**
Removes leading and/or trailing occurrences of a pattern within a character string.

**UPCASE**
Converts a character string to uppercase.

**Variable Length Character Functions**
The following functions manipulate variable length alphanumeric fields or character strings. For details, see *Variable Length Character Functions* on page 127.

**LENV**
Returns the actual length of an AnV field or the size of an An field.

**LOCASV**
Converts alphanumeric text to lowercase in an AnV field.

**POSITV**
Finds the starting position of a substring in an AnV field.

**SUBSTV**
Extracts a substring based on where it begins and its length in the parent string in an AnV field.

**TRIMV**
Removes leading and/or trailing occurrences of a pattern within a character string in an AnV field.

**UPCASV**
Converts a character string to uppercase in an AnV field.
**Character Functions for DBCS Code Pages**

The following functions manipulate character strings for DBCS code pages. For details, see *Character Functions for DBCS Code Pages* on page 139.

**DCTRAN**

Translates a single-byte or double-byte character to another character.

**DEDIT**

Extracts characters from or adds characters to a string.

**DSTRIP**

Removes a single-byte or double-byte character from a string.

**DSUBSTR**

Extracts a substring based on its length and position in the source string.

**JPTRANS**

Converts Japanese specific characters.

**Maintain-specific Character Functions**

The following functions manipulate alphanumeric fields or character strings. They are available only in the Maintain language. For details, see *Maintain-specific Character Functions* on page 151.

**CHAR2INT**

Translates an ASCII or EBCDIC character to the integer value it represents, depending on the operating system.

**INT2CHAR**

Translates an integer into the equivalent ASCII or EBCDIC character, depending on the operating system.

**LCWORD and LCWORD2**

Converts the letters in a character string to mixed case.

**LENGTH**

Measures the length of a character string, including trailing blanks.

**LJUST**

Left-justifies a character string within a field.
Types of Functions

**LOWER**
Converts a character string to lowercase.

**MASK**
Extracts characters from or adds characters to a character string.

**MNTGETTOK**
Divides a character string into substrings, called tokens.

**NLSCHR**
Converts a character from the native English code page to the running code page.

**OVRLAY**
Overlays a base character string with a substring.

**POSIT**
Finds the starting position of a substring within a larger string.

**RJUST**
Right-justifies a character string.

**SELECTS**
Decodes a value from a stack.

**STRAN**
Substitutes a substring for another substring in a character string.

**STRCMP**
Compares two alphanumeric strings using the ASCII or EBCDIC collating sequence.

**STRICMP**
Compares two alphanumeric strings using the ASCII or EBCDIC collating sequence, but ignoring case differences.

**STRNCMP**
Compares a specified number of characters in two character strings starting at the beginning of the strings using the EBCDIC or ASCII collating sequence.

**SUBSTR**
Extracts a substring based on where it begins and its length in the parent string.

**TRIM**
Removes trailing occurrences of a pattern within a character string.
TRIMLEN
Determines the length of a character string excluding trailing spaces.

UPCASE
Converts a character string to uppercase.

Data Source and Decoding Functions
The following functions search for data source records, retrieve data source records or values, and assign values. For details, see Data Source and Decoding Functions on page 177.

DB_LOOKUP
Retrieves a data value from a lookup data source.

DECODE
Assigns values based on the coded value of an input field.

FIND
Determines if an incoming data value is in an indexed FOCUS data source field.

LAST
Retrieves the preceding value for a field.

LOOKUP
Retrieves a data value from a cross-referenced FOCUS data source in a MODIFY request.
Available Languages: MODIFY, Maintain

Date Functions

In this section:
Standard Date Functions
Legacy Date Functions

The following functions manipulate dates. For details see Date Functions on page 199.

Standard Date Functions

DATEADD
Adds a unit to or subtracts a unit from a date format.
**Types of Functions**

**DATECVT**
Converts date formats.

**DATEDIF**
Returns the difference between two dates in units.

**DATEMOV**
Moves a date to a significant point on the calendar.

**DATETRAN**
Formats dates in international formats.

**DPART**
Extracts a component from a date field and returns it in numeric format.

**FIYR**
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.

**FIQTR**
Returns the financial quarter corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.

**FIYYQ**
Returns a financial date containing both the financial year and quarter that corresponds to a given calendar date.

**HMASK**
Extracts components from a date-time value and moves them to a target date-time field with all other components of the target field preserved.

**TODAY**
Retrieves the current date from the system.

**Legacy Date Functions**

**AYM**
Adds or subtracts months from dates that are in year-month format.

**AYMD**
Adds or subtracts days from dates that are in year-month-day format.
CHGDAT
Rearranges the year, month, and day portions of alphanumeric dates, and converts dates between long and short date formats.

DA
Convert dates to the corresponding number of days elapsed since December 31, 1899.
DADMY converts dates in day-month-year format.
DADYM converts dates in day-month-year format.
DAMDY converts dates in month-day-year format.
DAMYD converts dates in month-year-day format.
DAYDM converts dates in year-day-month format.
DAYMD converts dates in year-month-day format.

DMY, MDY, and YMD
Calculate the difference between two dates.

DOWK and DOWKL
Find the day of the week that corresponds to a date.

DT
Converts the number of days elapsed since December 31, 1899 to the corresponding date.
DTDMY converts numbers to day-month-year dates.
DTDYM converts numbers to day-year-month dates.
DTMDY converts numbers to month-day-year dates.
DTMYD converts numbers to month-year-day dates.
DTYDM converts numbers to year-day-month dates.
DTYMD converts numbers to year-month-day dates.

GREGDT
Converts dates in Julian format to year-month-day format.

JULDAT
Converts dates from year-month-day format to Julian (year-day format).
**YM**

Calculates the number of months that elapse between two dates. The dates must be in year-month format.

**Date-Time Functions**

The following functions manipulate date-time values. For details see *Date-Time Functions* on page 263.

**HADD**

Increments a date-time field by a given number of units.

**HCNVRT**

Converts a date-time field to a character string.

**HDATE**

Extracts the date portion of a date-time field, converts it to a date format, and returns the result in the format YYMD.

**HDIFF**

Calculates the number of units between two date-time values.

**HDTTM**

Converts a date field to a date-time field. The time portion is set to midnight.

**HEXTR**

Extracts components from a date-time value and moves them to a target date-time field with all other components set to zero.

**HGETC**

Stores the current date and time in a date-time field.

**HMASK**

Extracts components from a date-time value and moves them to a target date-time field with all other components of the target field preserved.

**HHMMSS**

Retrieves the current time from the system.

**HINPUT**

Converts an alphanumeric string to a date-time value.
**HMDNT**
Changes the time portion of a date-time field to midnight (all zeros).

**HNAME**
Extracts a specified component from a date-time field and returns it in alphanumeric format.

**HPART**
Extracts a specified component from a date-time field and returns it in numeric format.

**HSETPT**
Inserts the numeric value of a specified component into a date-time field.

**HTIME**
Converts the time portion of a date-time field to the number of milliseconds or microseconds.

**HTMTOTS/TIMETOTS**
Converts a time to a timestamp.

**Maintain-specific Date and Time Functions**

**In this section:**
- Maintain-specific Standard Date and Time Functions
- Maintain-specific Legacy Date Functions

The following functions manipulate dates and times. They are available only in the Maintain language. For details, see *Maintain-specific Date and Time Functions* on page 305.

**Maintain-specific Standard Date and Time Functions**

**HHMMSS**
Retrieves the current time from the system.

**Initial_HHMMSS**
Retrieves the time that the Maintain module was started.

**Initial_TODAY**
Retrieves the date that the Maintain module was started.
**TODAY**
Retrieves the current date from the system.

**TODAY2**
Retrieves the current date from the system.

**Maintain-specific Legacy Date Functions**

**ADD**
Adds a given number of days to a date.

**DAY**
Extracts the day of the month from a date.

**JULIAN**
Determines the number of days that have elapsed so far in the year up to a given date.

**MONTH**
Extracts the month from a date.

**QUARTER**
Determines the quarter of the year in which a date resides.

**SETMDY**
Sets a value to a date.

**SUB**
Subtracts a given number of days from a date.

**WEEKDAY**
Determines the day of the week for a date.

**YEAR**
Extracts the year from a date.

**Format Conversion Functions**

The following functions convert fields from one format to another. For details, see Format Conversion Functions on page 317

**ATODBL**
Converts a number in alphanumeric format to double-precision format.
EDIT
Converts an alphanumeric field that contains numeric characters to numeric format or converts a numeric field to alphanumeric format.

FPRINT
Converts a field to alphanumeric format.

FTOA
Converts a number in a numeric format to alphanumeric format.

HEXBYT
Obtains the ASCII or EBCDIC character equivalent of a decimal integer value.

ITONUM
Converts a large binary integer in a non-FOCUS data source to double-precision format.

ITOPACK
Converts a large binary integer in a non-FOCUS data source to packed-decimal format.

ITOZ
Converts a number in numeric format to zoned format.

PCKOUT
Writes a packed number of variable length to an extract file.

PTOA
Converts a packed decimal number from numeric format to alphanumeric format.

UFMT
Converts characters in alphanumeric field values to hexadecimal representation.

XTPACK
Stores a packed number with up to 31 significant digits in an alphanumeric field, retaining decimal data.

Numeric Functions
The following functions perform calculations on numeric constants or fields. For details, see "Numeric Functions" on page 343.

ABS
Returns the absolute value of a number.
Types of Functions

**ASIS**
Distinguishes between a blank and a zero in Dialogue Manager.

**BAR**
Produces a horizontal bar chart.

**CHKPCK**
Validates the data in a field described as packed format.

**DMOD, FMOD, and IMOD**
Calculate the remainder from a division.

**EXP**
 Raises the number "e" to a specified power.

**EXPN**
Is an operator that evaluates a number expressed in scientific notation. For information, see *Using Expressions* in the *Creating Reports* manual.

**FMLINFO**
Returns the FOR value associated with each row in an FML report.

**FMLLIST**
Returns a string containing the complete tag list for each row in an FML request.

**FMLFOR**
Retrieves the tag value associated with each row in an FML request.

**FMLCAP**
Retrieves the caption value for each row in an FML hierarchy request.

**INT**
Returns the integer component of a number.

**LOG**
Returns the natural logarithm of a number.

**MAX and MIN**
Return the maximum or minimum value, respectively, from a list of values.

**MIRR**
Calculates the modified internal rate of return for a series of periodic cash flows.
**NORMSDST and NORMSINV**

Perform calculations on a standard normal distribution curve.

**PRDNOR and PRDUNI**

Generate reproducible random numbers.

**RDNORM and RDUNIF**

Generate random numbers.

**SQRT**

Calculates the square root of a number.

**XIRR**

Calculates the internal rate of return for a series of cash flows that can be periodic or non-periodic.

### System Functions

The following functions call the operating system to obtain information about the operating environment or to use a system service. For details, see *System Functions* on page 379.

**CLSDREC**

Closes a file and frees the memory used to store information about open files.

**FEXERR**

Retrieves an Information Builders error message.

**FINDMEM**

Determines if a specific member of a partitioned data set (PDS) exists in batch processing.

Available Operating Systems: z/OS

**GETPDS**

Determines if a specific member of a partitioned data set (PDS) exists, and if it does, returns the PDS name.

Available Operating Systems: z/OS

**GETUSER**

Retrieves the ID of the connected user.

**MVSDYNAM**

Transfers a FOCUS DYNAM command to the DYNAM command processor.
Available Operating Systems: z/OS

**PUTDDREC**

Writes a character string as a record in a sequential file. Opens the file if it is closed.

**SLEEP**

Suspends execution for a specified number of seconds.

**SYSVAR**

Retrieves the Value of a z/OS System Variable.

Available Operating Systems: z/OS

### 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></td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>37</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>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</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>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
<td>semicolon</td>
</tr>
<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>Decimal</td>
<td>ASCII</td>
<td>ASCII</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------</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>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>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
<td>!</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
<td>$</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
<td>*</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
<td>)</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
<td>;</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
<td>¬</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>-</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>/</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>g</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>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</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>126</td>
<td>~</td>
<td>=</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>&quot;</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>
</tbody>
</table>
# 1. Introducing Functions

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<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>164</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td>165</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>166</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>167</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>168</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>169</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>185</td>
<td>`</td>
<td>grave accent</td>
</tr>
<tr>
<td>193</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Decimal</td>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>194</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>195</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>196</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>197</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>198</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>199</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>200</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>201</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>209</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>210</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>211</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>212</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>213</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>214</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>215</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>216</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>217</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>226</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>227</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>228</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>229</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>230</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>
## 1. Introducing Functions

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<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>
The following topics describe the considerations for supplying arguments in a function, and explain how to use a function in a command and access functions stored externally.

**Note:** FOCUS is fully LE compliant, and all FOCUS applications must be LE compliant.
Calling a Function

How to:

Call a Function
Store Output in a Field
Access the Maintain MNTUWS Function Library

You can call a function from a COMPUTE, DEFINE, or VALIDATE command. You can also call functions from a Dialogue Manager command, a Financial Modeling Language (FML) command, or a Maintain command. A function is called with the function name, arguments, and, for external functions, an output field.

For more information on external functions, see Types of Functions on page 19.

Some Maintain-specific functions require that the MNTUWS function library be retrieved when calling the function. For functions that require this, it is specified in the detailed information for that function. For more information on retrieving the MNTUWS library, see How to Access the Maintain MNTUWS Function Library on page 46.

Syntax: How to Call a Function

function(arg1, arg2, ... [outfield])

where:

function
  Is the name of the function.

arg1, arg2, ...
  Are the arguments.

outfield
  Is the field that contains the result, or the format of the output value enclosed in single quotation marks. This argument is required only for external functions.

In Dialogue Manager, you must specify the format. In Maintain, you must specify the name of the field.
**Syntax:**

How to Store Output in a Field

```
COMPUTE field(fmt) = function(input1, input2,... [outfield]);
```
or

```
DEFINE FILE file
field(fmt) = function(input1, input2,... [outfield]);
```
or

```
-SET &var = function(input1, input2,... [outfield]);
```

where:

- **DEFINE**
  - Creates a virtual field that may be used in a request as though it is a real data source field.

- **COMPUTE**
  - Calculates one or more temporary fields in a request. The field is calculated after all records have been selected, sorted, and summed.

- **field**
  - Is the field that contains the result.

- **file**
  - Is the file in which the virtual field is created.

- **var**
  - Is the variable that contains the result.

- **fmt**
  - Is the format of the field that contains the result.

- **function**
  - Is the name of the function, up to eight characters long.

- **input1, input2,...**
  - Are the input arguments, which are data values or fields used in function processing. For more information about arguments, see Supplying an Argument in a Function on page 46.

- **outfield**
  - Is the field that contains the result, or the format of the output value enclosed in single quotation marks. This argument is required only for external functions.

In Dialogue Manager, you must specify the format. In Maintain, you must specify the name of the field.
How to Access the Maintain MNTUWS Function Library

Place the following statement directly after the MAINTAIN command at the top of your procedure:

```
MODULE IMPORT (MNTUWS);
```

Supplying an Argument in a Function

In this section:
- Argument Types
- Argument Formats
- Argument Length
- Number and Order of Arguments
- Verifying Function Parameters

When supplying an argument in a function, you must understand which types of arguments are acceptable, the formats and lengths for these arguments, and the number and order of these arguments.

Argument Types

The following are acceptable arguments for a function:

- Numeric constant, such as 6 or 15.
- Date constant, such as 022802.
- Date in alphanumeric, numeric, date, or AnV format.
- Alphanumeric literal, such as STEVENS or NEW YORK NY. A literal must be enclosed in single quotation marks.
- Number in alphanumeric format.
- Field name, such as FIRST_NAME or HIRE_DATE. A field can be a data source field or temporary field. The field name can be up to 66 characters long or a qualified field name, unique truncation, or alias.
Expression, such as a numeric, date, or alphanumeric expression. An expression can use arithmetic operators and the concatenation sign (|). For example, the following are valid expressions:

```
CURR_SAL * 1.03
and
FN || LN
```

- Dialogue Manager variable, such as &CODE or &DDNAME.
- Format of the output value enclosed in single quotation marks.
- Another function.
- Label or other row or column reference (such as R or E), or name of another RECAP calculation, when the function is called in an FML RECAP command.

### Argument Formats

Depending on the function, an argument can be in alphanumeric, numeric, or date format. If you supply an argument in the wrong format, you will cause an error or the function will not return correct data. The following are the types of argument formats:

- **Alphanumeric argument.** An alphanumeric argument is stored internally as one character per byte. An alphanumeric argument can be a literal, an alphanumeric field, a number or date stored in alphanumeric format, an alphanumeric expression, or the format of an alphanumeric field. A literal is enclosed in single quotation marks, except when specified in operating systems that support Dialogue Manager RUN commands (for example, -MVS RUN).

- **Numeric argument.** A numeric argument is stored internally as a binary or packed number. A numeric argument includes integer (I), floating-point single-precision (F), floating-point double-precision (D), and packed decimal (P) formats. A numeric argument can be a numeric constant, field, or expression, or the format of a numeric field.

  All numeric arguments are converted to floating-point double-precision format when used with a function, but results are returned in the format specified for the output field.

- **Date argument.** A date argument can be in either alphanumeric, numeric, or date format. The list of arguments for the individual function will specify what type of format the function accepts. A date argument can be a date in alphanumeric, numeric, or date format; a date field or expression; or the format of a date field.

  If you supply an argument with a two-digit year, the function assigns a century based on the DATEFNS, YRTHRESH, and DEFCENT parameter settings.
Argument Length

An argument is passed to a function by reference, meaning that the memory location of the argument is passed. No indication of the length of the argument is given.

You must supply the argument length for alphanumeric strings. Some functions require a length for the input and output arguments (for example, SUBSTR), and others use one length for both arguments (for example, UPCASE).

Be careful to ensure that all lengths are correct. Providing an incorrect length can cause incorrect results:

- If the specified length is shorter than the actual length, a subset of the string is used. For example, passing the argument ' ABCDEF ' and specifying a length of 3 causes the function to process a string of 'ABC'.

- If the specified length is too long, whatever is in memory up to that length is included. For example, passing an argument of ' ABC ' and specifying a length of 6 causes the function to process a string beginning with 'ABC' plus the three characters in the next three positions of memory. Depending on memory utilization, the extra three characters could be anything.

Some operating system routines are very sensitive to incorrectly specified lengths and read them into incorrectly formatted memory areas.

Number and Order of Arguments

The number of arguments required varies according to each function. Functions supplied by Information Builders may require up to six arguments. User-written subroutines may require a maximum of 200 arguments including the output argument. If a function requires more than 200 arguments, you must use two or more calls to pass the arguments to the function.

Arguments must be specified in the order shown in the syntax of each function. The required order varies according to the function.

Verifying Function Parameters

**How to:**

Enable Parameter Verification
Control Function Parameter Verification

The USERFCHK setting controls the level of verification applied to DEFINE FUNCTION and Information Builders-supplied function arguments. It does not affect verification of the number of parameters; the correct number must always be supplied.

USERFCHK is not supported from Maintain.
Functions typically expect parameters to be a specific type or have a length that depends on the value of another parameter. It is possible in some situations to enforce these rules by truncating the length of a parameter and, therefore, avoid generating an error at run time.

The level of verification and possible conversion to a valid format performed depends on the specific function. The following two situations can usually be converted satisfactorily:

- If a numeric parameter specifies a maximum size for an alphanumeric parameter, but the alphanumeric string supplied is longer than the specified size, the string can be truncated.
- If a parameter supplied as a numeric literal specifies a value larger than the maximum size for a parameter, it can be reduced to the proper value.

**Syntax: How to Enable Parameter Verification**

Parameter verification can be enabled only for DEFINE FUNCTIONs and functions supplied by Information Builders. If your site has a locally written function with the same name as an Information Builders-supplied function, the USERFNS setting determines which function is used.

```
SET USERFNS= {SYSTEM | LOCAL}
```

where:

- **SYSTEM**
  
  Gives precedence to functions supplied by Information Builders. SYSTEM is the default value. This setting is required in order to enable parameter verification.

- **LOCAL**
  
  Gives precedence to locally written functions. Parameter verification is not performed with this setting in effect.

**Note:** When USERFNS is set to LOCAL, DT functions only display a six-digit date.
**Syntax:** How to Control Function Parameter Verification

Issue the following command in FOCPARM, FOCPROF, on the command line, in a FOCEXEC, or in an ON TABLE command. Note that the USERFNS=SYSTEM setting must be in effect.

```
SET USERFCHK = setting
```

where:

```
setting
```

Can be one of the following:

- **ON** is the default value. Verifies parameters in requests, but does not verify parameters for functions used in Master File DEFINEs. If a parameter has an incorrect length, an attempt is made to fix the problem. If such a problem cannot be fixed, an error message is generated and the evaluation of the affected expression is terminated.

- Because parameters are not verified for functions specified in a Master File, no errors are reported for those functions until the DEFINE field is used in a subsequent request when, if a problem occurs, the following message is generated:

  `(FOC003) THE FIELDNAME IS NOT RECOGNIZED`

- **OFF** does not verify parameters except in the following cases:
  - If a parameter that is too long would overwrite the memory area in which the computational code is stored, the size is automatically reduced without issuing a message.
  - If an alphanumeric parameter is too short, it is padded with blanks to the correct length.

  **Note:** The OFF setting will be deprecated in a future release.

- **FULL** is the same as ON, but also verifies parameters for functions used in Master File DEFINEs.

- **ALERT** verifies parameters in a request without halting execution when a problem is detected. It does not verify parameters for functions used in Master File DEFINEs. If a parameter has an incorrect length and an attempt is made to fix the problem behind the scenes, the problem is corrected with no message. If such a problem cannot be fixed, a warning message is generated. Execution then continues as though the setting were OFF, but the results may be incorrect.

  **Note:** If a parameter provided is the incorrect type, verification fails and processing terminates.
Example: Verifying Parameters With Correctable Errors

The following request uses SUBSTR to extract the substring that starts in position 6 and ends in position 14 of the TITLE field. The fifth argument specifies a substring length (500) that is too long (it should be no longer than 9):

```
SET USERFCHK = ON
TABLE FILE MOVIES
PRINT TITLE
COMPUTE
  NEWTITLE/A9  = SUBSTR(39, TITLE, 6  ,14, 500, NEWTITLE);
WHERE CATEGORY EQ 'CHILDREN'
END
```

When the request is executed with USERFCHK=ON or OFF, the incorrect length is corrected and the request continues processing:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>NEWTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>S, THE</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>Y DOG, TH</td>
</tr>
<tr>
<td>SCOOBY-DOO-A DOG IN THE RUFF</td>
<td>Y-DOO-A D</td>
</tr>
<tr>
<td>ALICE IN WONDERLAND</td>
<td>IN WONDE</td>
</tr>
<tr>
<td>SESAME STREET-BEDTIME STORIES AND SONGS</td>
<td>E STREET-</td>
</tr>
<tr>
<td>ROMPER ROOM-ASK MISS MOLLY</td>
<td>R ROOM-AS</td>
</tr>
<tr>
<td>SLEEPING BEAUTY</td>
<td>ING BEAUT</td>
</tr>
<tr>
<td>BAMBI</td>
<td></td>
</tr>
</tbody>
</table>

Example: Verifying Parameters With Uncorrectable Errors

The following request has an incorrect data type in the last argument to SUBSTR. This parameter should specify an alphanumeric field or format for the extracted substring:

```
SET USERFCHK = ON
TABLE FILE MOVIES
PRINT TITLE
COMPUTE
  NEWTITLE/F9  = SUBSTR(39, TITLE, 6  ,14, 500, 'F9');
WHERE CATEGORY EQ 'CHILDREN'
END
```

When the request is executed with USERFCHK=ON, a message is produced and the request terminates:

```
ERROR AT OR NEAR LINE 5 IN PROCEDURE USERFC3 FOCEXEC
(FOC279) NUMERIC ARGUMENTS IN PLACE WHERE ALPHA ARE CALLED FOR
(FOC009) INCOMPLETE REQUEST STATEMENT
UNKNOWN FOCUS COMMAND WHERE
BYPASSING TO END OF COMMAND
```
When the request is executed with USERFCHK=OFF, no verification is done and no message is produced. The request executes and produces incorrect results. In some environments, this type of error may cause abnormal termination of the application:

<table>
<thead>
<tr>
<th>DIRECTOR</th>
<th>TITLE</th>
<th>NEWTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARTON C.</td>
<td>SMURFS, THE</td>
<td>**********</td>
</tr>
<tr>
<td>SCOOBY-DOO-A DOG IN THE RUFF</td>
<td>**********</td>
<td></td>
</tr>
<tr>
<td>GEROMINI</td>
<td>ALICE IN WONDERLAND</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SESAME STREET-BEDTIME STORIES AND SONGS -265774</td>
<td></td>
</tr>
<tr>
<td>ROMPER ROOM-ASK MISS MOLLY</td>
<td>**********</td>
<td></td>
</tr>
<tr>
<td>DISNEY W.</td>
<td>SLEEPING BEAUTY</td>
<td>**********</td>
</tr>
<tr>
<td>DISNEY W.</td>
<td>BAMBI</td>
<td>0</td>
</tr>
</tbody>
</table>

Calling a Function From a DEFINE, COMPUTE, or VALIDATE Command

How to:
Call a Function From a COMPUTE, DEFINE, or VALIDATE Command

You can call a function from a DEFINE command or Master File attribute, a COMPUTE command, or a VALIDATE command.

Syntax:
How to Call a Function From a COMPUTE, DEFINE, or VALIDATE Command

```
DEFINE [FILE filename]
  tempfield[/format] = function(input1, input2, input3, ... [outfield]);
COMPUTE
  tempfield[/format] = function(input1, input2, input3, ... [outfield]);
VALIDATE
  tempfield[/format] = function(input1, input2, input3, ... [outfield]);
```

where:

- **filename**
  
  Is the data source being used.

- **tempfield**
  
  Is the temporary field created by the DEFINE or COMPUTE command. This is the same field specified in `outfield`. If the function call supplies the format of the output value in `outfield`, the format of the temporary field must match the `outfield` argument.

- **format**
  
  Is the format of the temporary field. The format is required if it is the first time the field is created; otherwise, it is optional. The default value is D12.2.
function
    Is the name of the function.

input1, input2, input3...
    Are the arguments.

outfield
    Is the field that contains the result, or the format of the output value enclosed in single quotation marks. This is required only for external functions.
    In Dialogue Manager, you must specify the format. In Maintain, you must specify the name of the field.

Calling a Function From a Dialogue Manager Command

You can call a function with Dialogue Manager in the following ways:

- From a -SET command, storing the result of a function in a variable. For more information, see Assigning the Result of a Function to a Variable on page 54.

- From an -IF command. For more information, see Calling a Function in WHERE or IF Criteria on page 59.

- From an operating system -RUN command. For more information, see Calling a Function From an Operating System RUN Command on page 57.

Dialogue Manager converts a numeric argument to double-precision format. This occurs when the value of the argument is numeric; this is not affected by the format expected by the function. This means you must be careful when supplying arguments for a function in Dialogue Manager.

If the function expects an alphanumeric string and the input is a numeric string, incorrect results will occur because of conversion to floating-point double-precision. To resolve this problem, append a non-numeric character to the end of the string, but do not count this extra character in the length of the argument.
Dialogue Manager date variables such as &YYMD return alphanumeric legacy dates, not a date format (an offset from a base date). If a function requires a date offset rather than a legacy date, you must convert any date variable to a date offset (using the DATECVT function) before using it as an argument. You can then convert the result back to a legacy date, again with the DATECVT function. For example:

-SET &TODAY_OFFSET=DATECVT(&YYMD, 'I8YYMD', 'YYMD');
-SET &BEG_CUR_YR=DATEMOV(&TODAY_OFFSET.EVAL, 'BOY');
-SET &CLOSE_DTBOY=DATECVT(&BEG_CUR_YR.EVAL, 'YYMD', 'I8YYMD');

### Assigning the Result of a Function to a Variable

#### How to:
Assign the Result of a Function to a Variable

You can store the result of a function in a variable with the -SET command.

A Dialogue Manager variable contains only alphanumeric data. If a function returns a numeric value to a Dialogue Manager variable, the value is truncated to an integer and converted to alphanumeric format before being stored in the variable.

#### Syntax:
**How to Assign the Result of a Function to a Variable**

-SET &variable = function(arg1, arg2[.LENGTH],..., 'format');

where:

- **variable**
  - Is the variable to which the result will be assigned.

- **function**
  - Is the function.

- **arg1, arg2**
  - Are the function's arguments.

- **.LENGTH**
  - Returns the length of the variable. If a function requires the length of a character string as an input argument, you can prompt for the character string and determine the length with the .LENGTH suffix.

- **format**
  - Is the format of the result enclosed in single quotation marks. You cannot specify a Dialogue Manager variable for the output argument unless you use the .EVAL suffix; however, you can specify a variable for an input argument.
**Example:**  **Calling a Function From a -SET Command**

AYMD adds 14 days to the value of &INDATE. The &INDATE variable is previously set in the procedure in the six-digit year-month-day format.

```
-SET &OUTDATE = AYMD(&INDATE, 14, 'I6');
```

The format of the output date is a six-digit integer (I6). Although the format indicates that the output is an integer, it is stored in the &OUTDATE variable as a character string. For this reason, if you display the value of &OUTDATE, you will not see slashes separating the year, month, and day.

**Branching Based on the Result of a Function**

**How to:**

Branch Based on the Result of a Function

You can branch based on the result of a function by calling a function from a Dialogue Manager -IF command.

If a branching command spans more than one line, continue it on the next line by placing a dash (-) in the first column.

**Syntax:**  **How to Branch Based on the Result of a Function**

```
-IF function(args) relation expression GOTO label1 [ELSE GOTO label2];
```

where:

- **function**
  - Is the function.

- **args**
  - Are the arguments.

- **relation**
  - Is an operator that determines the relationship between the function and expression, for example, EQ or LE.

- **expression**
  - Is a value, logical expression, or function. Do not enclose a literal in single quotation marks unless it contains a comma or embedded blank.
**Example:** Branching Based on the Result of a Function

The result of the AYMD function provides a condition for a -IF test. One of two requests is executed, depending on the result of the function:

-LOOP

1. -PROMPT &INDATE.ENTER START DATE IN YEAR-MONTH-DAY FORMAT OR ZERO TO EXIT:. 
2. IF &INDATE EQ 0 GOTO EXIT; 
3. SET &WEEKDAY = DOWK(&INDATE, 'A4'); 
4. -TYPE START DATE IS &WEEKDAY &INDATE 
5. -PROMPT &DAYS.ENTER ESTIMATED PROJECT LENGTH IN DAYS:. 
6. -IF AYMD(&INDATE, &DAYS, 'I6YMD') LT 960101 GOTO EARLY; 
7. -TYPE LONG PROJECT 
   **-**EX LONGPROJ 
   -RUN 
   -GOTO EXIT 
8. -EARLY 
   -TYPE SHORT PROJECT 
   **-**EX SHRTPROJ 
   -RUN 
   -GOTO EXIT 
   -EXIT 

The procedure processes as follows:

1. It prompts for the start date of a project in YYMMDD format. 
2. If you enter a 0, it passes control to -EXIT which terminates execution. 
3. The DOWK function obtains the day of the week for the start date. 
4. The -TYPE command displays the day of the week and start date of the project. 
5. The procedure prompts for the estimated length of the project in days. 
6. The AYMD function calculates the date that the project will finish. If this date is before January 1, 1996, the -IF command branches to the label EARLY.
7. If the project will finish on or after January 1, 1996, the TYPE command displays the words LONG PROJECT and exits.

8. If the procedure branches to the label EARLY, the TYPE command displays the words SHORT PROJECT and exits.

**Calling a Function From an Operating System RUN Command**

**How to:**

Call a Function From an Operating System -RUN Command

You can call a function that contains only alphanumeric arguments from a Dialogue Manager -TSO RUN or -MVS RUN command. This type of function performs a specific task but typically does not return a value.

If a function requires an argument in numeric format, you must first convert it to floating-point double-precision format using the ATODBL function because, unlike the -SET command, an operating system RUN command does not automatically convert a numeric argument to double-precision.

**Syntax:**

How to Call a Function From an Operating System -RUN Command

```
{TSO|MVS} RUN function, input1, input2, ... [, &output]
```

where:

- **TSO** | **MVS**
  - Is the operating system.

- **function**
  - Is the name of the function.

- **input1, input2, ...**
  - Are the arguments. Separate the function name and each argument with a comma. Do not enclose an alphanumeric literal in single quotation marks. If a function requires the length of a character string as an argument, you can prompt for the character string, then use the .LENGTH suffix to test the length.
&output
Is a Dialogue Manager variable. Include this argument if the function returns a value; otherwise, omit it. If you specify an output variable, you must pre-define its length using a -SET command.

For example, if the function returns a value that is eight bytes long, define the variable with eight characters enclosed in single quotation marks before the function call:

-SET &output = '12345678';

**Example:** **Calling a Function From an Operating System -RUN Command**

The following example calls the CHGDAT function from a -MVS RUN command:

-SET &RESULT = '12345678901234567';
-MVS RUN CHGDAT, YYMD., MXDYY, &YYMD, &RESULT
-TYPE &RESULT

**Calling a Function From Another Function**

**How to:**
Call a Function From Another Function

A function can be an argument for another function.

**Syntax:** **How to Call a Function From Another Function**

field = function([arguments,] function2[arguments2,] arguments);

where:

- **field**
  Is the field that contains the result of the function.

- **function**
  Is a function.

- **arguments**
  Are arguments for function.

- **function2**
  Is the function that is an argument for function.

- **arguments2**
  Are arguments for function2.
**Example:** Calling a Function From Another Function

In the following example, the AYMD function is an argument for the YMD function:

```
-SET &DIFF = YMD(&YYMD, AYMD(&YYMD, 4, 'I8'));
```

**Calling a Function in WHERE or IF Criteria**

**In this section:**
Using a Calculation or Compound IF Command

**How to:**
Call a Function in WHERE Criteria
Call a Function in IF Criteria

You can call a function in WHERE or IF criteria. When you do this, the output value of the function is compared against a test value.

**Syntax:** How to Call a Function in WHERE Criteria

```
WHERE function relation expression
```

where:

- **function**
  Is a function.

- **relation**
  Is an operator that determines the relationship between the function and expression, for example, EQ or LE.

- **expression**
  Is a constant, field, or function. A literal must be enclosed in single quotation marks.

**Syntax:** How to Call a Function in IF Criteria

```
IF function relation value
```

where:

- **function**
  Is a function.
**relation**

Is an operator that determines the relationship between the function and expression, for example, EQ or LE.

**value**

Is a constant. In a DEFINE or COMPUTE command, the value must be enclosed in single quotation marks.

### Example: Calling a Function in WHERE Criteria

The SUBSTR function extracts the first two characters of LAST_NAME as a substring, and the request prints an employee's name and salary if the substring is MC.

```
TABLE FILE EMPLOYEE
PRINT FIRST_NAME LAST_NAME CURR_SAL
WHERE SUBSTR(15, LAST_NAME, 1, 2, 2, 'A2') IS 'MC';
END
```

The output is:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>CURR_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>MCCOY</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>ROGER</td>
<td>MCKNIGHT</td>
<td>$16,100.00</td>
</tr>
</tbody>
</table>

### Using a Calculation or Compound IF Command

You must specify the format of the output value in a calculation or compound IF command. There are two ways to do this:

- Pre-define the format within a separate command. In the following example, the AMOUNT field is pre-defined with the format D8.2 and the function returns a value to the output field AMOUNT. The IF command tests the value of AMOUNT and stores the result in the calculated value, AMOUNT_FLAG.

```
COMPUTE
AMOUNT/D8.2 =;
AMOUNT_FLAG/A5 = IF function(input1, input2, AMOUNT) GE 500
    THEN 'LARGE' ELSE 'SMALL';
```

- Supply the format as the last argument in the function call. In the following example, the command tests the returned value directly. This is possible because the function defines the format of the returned value (D8.2).

```
DEFINE
AMOUNT_FLAG/A5 = IF function(input1, input2, 'D8.2') GE 500
    THEN 'LARGE' ELSE 'SMALL';
```
Calling a Function in WHEN Criteria

**How to:**

Call a Function in WHEN Criteria

You can call a function in WHEN criteria as part of a Boolean expression.

**Syntax:**

**How to Call a Function in WHEN Criteria**

WHEN({function|value} relation {function|value});

or

WHEN NOT(function)

where:

*function*

Is a function.

*value*

Is a value or logical expression.

*relation*

Is an operator that determines the relationship between the value and function, for example, LE or GT.

**Example:**

**Calling a Function in WHEN Criteria**

This request checks the values in LAST_NAME against the result of the CHKFMT function. When a match occurs, the request prints a sort footing.

```
TABLE FILE EMPLOYEE
PRINT DEPARTMENT BY LAST_NAME
ON LAST_NAME SUBFOOT
"*** LAST NAME <LAST_NAME DOES MATCH MASK"
WHEN NOT CHKFMT(15, LAST_NAME, 'SMITH          ', 'I6');
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>PRODUCTION</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>MIS</td>
</tr>
<tr>
<td>CROSS</td>
<td>MIS</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MIS</td>
</tr>
<tr>
<td>IRVING</td>
<td>PRODUCTION</td>
</tr>
<tr>
<td>JONES</td>
<td>MIS</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MIS</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>PRODUCTION</td>
</tr>
<tr>
<td>ROMANS</td>
<td>PRODUCTION</td>
</tr>
<tr>
<td>SMITH</td>
<td>MIS</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td></td>
</tr>
</tbody>
</table>

*** LAST NAME SMITH DOES MATCH MASK

STEVENS  PRODUCTION

**Calling a Function From a RECAP Command**

**How to:**

Call a Function From a RECAP Command

You can call a function from an FML RECAP command.

**Syntax:**

**How to Call a Function From a RECAP Command**

```
RECAP name[(n) | (n,m) | (n,m,i)][/format1] = function(input1,...,['format2']);
```

where:

- **name**
  Is the name of the calculation.

- **n**
  Displays the value in the column number specified by \( n \). If you omit the column number, the value appears in all columns.

- **n, m**
  Displays the value in all columns beginning with the column number specified by \( n \) and ending with the column number specified by \( m \).

- **n, m, i**
  Displays the value in the columns beginning with the column number specified by \( n \) and ending with the column number specified by \( m \) by the interval specified by \( i \). For example, if \( n \) is 1, \( m \) is 5, and \( i \) is 2, the value displays in columns 1, 3, and 5.
format1

Is the format of the calculation. The default value is the format of the report column.

function

Is the function.

input1,...

Are the input arguments, which can include numeric constants, alphanumeric literals, row and column references (R notation, E notation, or labels), and names of other RECAP calculations.

format2

Is the format of the output value enclosed in single quotation marks. If the calculation's format is larger than the column width, the value appears in that column as asterisks.

Example: Calling a Function in a RECAP Command

This request sums the AMOUNT field for account 1010 using the label CASH, account 1020 using the label DEMAND, and account 1030 using the label TIME. The MAX function displays the maximum value of these accounts.

```
TABLE FILE LEDGER
SUM AMOUNT FOR ACCOUNT
1010 AS 'CASH ON HAND' LABEL CASH OVER
1020 AS 'DEMAND DEPOSITS' LABEL DEMAND OVER
1030 AS 'TIME DEPOSITS' LABEL TIME OVER
BAR OVER
RECAP MAXCASH = MAX(CASH, DEMAND, TIME); AS 'MAX CASH'
END
```

The output is:

```
AMOUNT
--------
CASH ON HAND      8,784
DEMAND DEPOSITS   4,494
TIME DEPOSITS     7,961
--------
MAX CASH          8,784
```
Storing and Accessing an External Function

Internal functions are built in and do not require additional work to access. External functions are stored in load libraries from which they must be retrieved. The way these external functions are accessed is determined by your platform. These techniques may not have to be used every time a function is accessed. Access to a load library may be set only once at the time of installation.

You can also access private user-written subroutines. If you have a private collection of subroutines (that is, you created your own or use customized subroutines), do not store them in the function library. Store them separately to avoid overwriting them whenever your site installs a new release. For more information on creating a subroutine, see Creating a Subroutine on page 399.

Storing and Accessing a Function on z/OS

How to:

Allocate a Load Library in z/OS Batch
Allocate a Load Library in TSO
Allocate a Load Library

On z/OS, load libraries are partitioned data sets containing link-edited modules. These libraries are stored as EDALIB.LOAD or FUSELIB.LOAD. In addition, your site may have private subroutine collections stored in separate load libraries. If so, you must allocate those libraries.

Procedure:  How to Allocate a Load Library in z/OS Batch

To use a function stored as a load library, allocate the load library to ddname USERLIB in your JCL or CLIST.

The search order is USERLIB, STEPLIB, JOBLIB, link pack area, and linklist.

Example:  Allocating the Load Library BIGLIB.LOAD in z/OS Batch (JCL)

//USERLIB DD DISP=SHR, DSN=BIGLIB.LOAD
**Procedure:** How to Allocate a Load Library in TSO

Allocate the load library to ddname USERLIB using the ALLOCATE command. You can issue the ALLOCATE command:

- In TSO before entering a FOCUS session.
- Before executing a request in a FOCUS session.
- In your PROFILE FOCEXEC.

If you are in a FOCUS session, you can also use the DYNAM ALLOCATE command.

If you are in a FOCUS session, you can also use the DYNAM ALLOCATE command.

**Syntax:** How to Allocate a Load Library

```plaintext
{MVS|TSO} ALLOCATE FILE(USERLIB) DSN(lib1 lib2 lib3 ...) SHR
```

or

```plaintext
DYNAM ALLOC FILE USERLIB DA lib SHR
```

where:

- **MVS|TSO**
  
  Is the prefix if you issue the ALLOCATE command from your application or include it in your PROFILE FOCEXEC.

- **USERLIB**
  
  Is the ddname to which you allocate a load library.

- **lib1 lib2 lib3...**
  
  Are the names of the load libraries, concatenated to ddname USERLIB.

**Example:** Allocating the FUSELIB.LOAD Load Library

```plaintext
TSO ALLOC FILE(USERLIB) DSN('MVS.FUSELIB.LOAD') SHR
```

or

```plaintext
DYNAM ALLOC FILE USERLIB DA MVS.FUSELIB.LOAD SHR
```
**Example:** **Concatenating a Load Library to USERLIB In TSO**

Suppose a report request calls two functions: BENEFIT stored in library SUBLIB.LOAD, and EXCHANGE stored in library BIGLIB.LOAD. To concatenate the BIGLIB and SUBLIB load libraries in the allocation for ddname USERLIB, issue the following commands:

```
DYNAM ALLOC FILE USERLIB DA SUBLIB.LOAD SHR
DYNAM ALLOC FILE BIGLIB DA BIGLIB.LOAD SHR
DYNAM CONCAT FILE USERLIB BIGLIB
```

The load libraries are searched in the order in which they are specified in the ALLOCATE command.

**Example:** **Concatenating a Load Library to STEPLIB in Batch (JCL)**

Concatenate the load library to the ddname STEPLIB in your JCL:

```
//FOCUS EXEC PGM=FOCUS
//STEPLIB     DD DSN=FOCUS.FOCLIB.LOAD,DISP=SHR
//            DD DSN=FOCUS.FUSELIB.LOAD,DISP=SHR
//.
//.
```

**Storing and Accessing a Function on UNIX**

No extra work is required.
Character Functions

Character functions manipulate alphanumeric fields and character strings.

Topics:
- Character Function Notes
- 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
- 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
- LJUST: Left-Justifying a String
- LOCASE: Converting Text to Lowercase
- 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
In addition to the functions discussed in this topic, there are character functions that are available only in the Maintain language. For information on these functions, see Maintain-specific Character Functions on page 151.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function on page 43.

ARGLEN: Measuring the Length of a String

**How to:**

Measure the Length of a Character String

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

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

**Syntax:**

**How to Measure the Length of a Character String**

ARGLEN(length, source_string, output)

where:

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

- **source_string**
  - Alphanumeric
  - Is the name of the field containing the character string.
output

Integer

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

Example: Measuring the Length of a Character String

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

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

The output is:

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

ASIS: Distinguishing Between Space and Zero

How to:
Distinguish Between a Space and a 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

```
ASIS(argument)
```

where:

- **argument**
  
  Alphanumeric

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

  - If you specify an alphanumeric 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.

```
-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.

```
-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
```
BITSON: Determining If a Bit Is On or Off

How to: Determine If a Bit Is On or Off

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

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

BITSON(bitnumber, source_string, output)

where:

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

source_string
  Alphanumeric
  Is the character string to be evaluated, enclosed in single quotation marks, or a field or variable that contains the character string. 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 and stores the result in BIT_24:

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
BIT_24/I1 = BITSON(24, LAST_NAME, BIT_24);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>BIT_24</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>1</td>
</tr>
<tr>
<td>JONES</td>
<td>1</td>
</tr>
<tr>
<td>MCCOY</td>
<td>1</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>1</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>1</td>
</tr>
<tr>
<td>CROSS</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**How to:**

Evaluate a Bit String

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

**Syntax:**  **How to Evaluate a Bit String**

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

where:

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

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

Integer

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

output

Integer

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

**Example: Evaluating a Bit String**

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

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

The output is:

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

**BYTVAL: Translating a Character to Decimal**

**How to:**

Translate a Character

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

**output**

Integer

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

**Example:**

**Translating the First Character of a Field**

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

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

The output on an ASCII platform is:

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

The output on an EBCDIC platform is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LAST_INIT_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>226</td>
</tr>
<tr>
<td>JONES</td>
<td>209</td>
</tr>
<tr>
<td>MCCOY</td>
<td>212</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>194</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>199</td>
</tr>
<tr>
<td>CROSS</td>
<td>195</td>
</tr>
</tbody>
</table>
Example: Returning the EBCDIC Value With Dialogue Manager

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

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

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

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

CHKFMT: Checking the Format of a String

How to: Check the Format of a Character String

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

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

Syntax: How to Check the Format of a Character String

CHKFMT(numchar, source_string, 'mask', output)

where:

numchar
- Integer
  - Is the number of characters being compared to the mask.
string
   Alphanumeric
   Is the character string to be checked enclosed in single quotation marks, or a field or
   variable that contains the character string.

'mask'
   Alphanumeric
   Is the mask, which contains the comparison characters enclosed in single quotation
   marks.
   Some characters in the mask are generic and represent character types. If a character
   in the string is compared to one of these characters and is the same type, it matches.
   Generic characters are:
   \( A \) is any letter between A and Z (uppercase or lowercase).
   \( 9 \) is any digit between 0–9.
   \( X \) is any letter between A–Z or any digit between 0-9.
   \( $ \) is any character.
   Any other character in the mask represents only that character. For example, if the third
   character in the mask is B, the third character in the string must be B to match.

output
   Integer
   Is the name of the field that contains the result, or the format of the output value
   enclosed in single quotation marks.
Checking the Format of a Field

**Example:**

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

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

The output is:

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

Checking the Format of a Field With MODIFY on z/OS

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

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

```
> EMPLOYEEFOCUS ON 12/05/96 AT 15.42.03
DATA FOR TRANSACTION 1
EMP_ID = 111w2
LAST_NAME = johnson
FIRST_NAME = greg
DEPARTMENT = production
BAD EMPLOYEE ID: 111W2
INVALID CHARACTER IN POSITION 4
DATA FOR TRANSACTION 2
EMP_ID = end
```

The procedure processes as follows:

1. The procedure searches the data source for the ID 111w2. If it does not find this ID, it continues processing the transaction.

2. CHKFMT checks the ID against the mask 99999, which represents five digits.

3. The fourth character in the ID, the letter w, is not a digit. The function returns the value 4 to the BAD_CHAR field.

4. The VALIDATE command tests the BAD_CHAR field. Since BAD_CHAR is not equal to 0, the procedure rejects the transaction and displays a message indicating the position of the invalid character in the ID.

### CTRAN: Translating One Character to Another

**How to:**

Translate One Character to Another

The CTRAN function translates a character within a character string to another character based on its decimal value. This function is especially useful for changing replacement characters to unavailable characters, or to characters that are difficult to input or unavailable on your keyboard. It can also be used for inputting characters that are difficult to enter when responding to a Dialogue Manager -PROMPT command, such as a comma or apostrophe. It eliminates the need to enclose entries in single quotation marks.
To use CTRAN, you 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 34.

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

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

**Syntax:**

**How to Translate One Character to Another**

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

where:

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

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

- **decimal**
  - Integer
  - 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
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**Example: Translating Spaces to Underscores on an ASCII Platform**

CTRAN translates the spaces in ADDRESS_LN3 (ASCII decimal value 32) to underscores (ASCII decimal value 95), and stores the result in ALT_ADDR:

```plaintext
TABLE FILE EMPLOYEE
PRINT ADDRESS_LN3 AND COMPUTE
ALT_ADDR/A20 = CTRAN(20, ADDRESS_LN3, 32, 95, ALT_ADDR);
BY EMP_ID
WHERE TYPE EQ 'HSM';
END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>ADDRESS_LN3</th>
<th>ALT_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>117593129</td>
<td>RUTHERFORD NJ 07073</td>
<td>RUTHERFORD_NJ_07073__</td>
</tr>
<tr>
<td>119265415</td>
<td>NEW YORK NY 10039</td>
<td>NEW_YORK_NY_10039___</td>
</tr>
<tr>
<td>119329144</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520____</td>
</tr>
<tr>
<td>123764317</td>
<td>NEW YORK NY 10001</td>
<td>NEW_YORK_NY_10001____</td>
</tr>
<tr>
<td>126724188</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520____</td>
</tr>
<tr>
<td>451123478</td>
<td>ROSELAND NJ 07068</td>
<td>ROSELAND_NJ_07068____</td>
</tr>
<tr>
<td>543729165</td>
<td>JERSEY CITY NJ 07300</td>
<td>JERSEY_CITY_NJ_07300</td>
</tr>
<tr>
<td>818692173</td>
<td>FLUSHING NY 11354</td>
<td>FLUSHING_NY_11354_____</td>
</tr>
</tbody>
</table>

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

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

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

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>ADDRESS_LN3</th>
<th>ALT_ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>117593129</td>
<td>RUTHERFORD NJ 07073</td>
<td>RUTHERFORD_NJ_07073__</td>
</tr>
<tr>
<td>119265415</td>
<td>NEW YORK NY 10039</td>
<td>NEW_YORK_NY_10039___</td>
</tr>
<tr>
<td>119329144</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520____</td>
</tr>
<tr>
<td>123764317</td>
<td>NEW YORK NY 10001</td>
<td>NEW_YORK_NY_10001____</td>
</tr>
<tr>
<td>126724188</td>
<td>FREEPORT NY 11520</td>
<td>FREEPORT_NY_11520____</td>
</tr>
<tr>
<td>451123478</td>
<td>ROSELAND NJ 07068</td>
<td>ROSELAND_NJ_07068____</td>
</tr>
<tr>
<td>543729165</td>
<td>JERSEY CITY NJ 07300</td>
<td>JERSEY_CITY_NJ_07300</td>
</tr>
<tr>
<td>818692173</td>
<td>FLUSHING NY 11354</td>
<td>FLUSHING_NY_11354_____</td>
</tr>
</tbody>
</table>
Example: Inserting Accented Letter E's With MODIFY

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

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

The request is:

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

A sample execution follows:

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

ENTER EMPLOYEE'S ID: 999888777

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

FIRST_NAME: AD*LE     LAST_NAME: MOLI*RE

ENTER THE DEPARTMENT ASSIGNMENT: SALES
```
The request processes as:

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

2. Enter the following data:
   
   EMPLOYEE ID: 999888777
   FIRST_NAME: AD*LE
   LAST_NAME: MOLI*RE
   DEPARTMENT: SALES

3. The procedure searches the data source for the employee ID. If it does not find it, it continues processing the request.

4. CTRAN converts the asterisks into È’s in both the first and last names (ADÈLE MOLIÈRE).

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

   ENTER EMPLOYEE'S ID:

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

   FIRST_NAME:              LAST_NAME:

   ENTER THE DEPARTMENT ASSIGNMENT:

   FIRST_NAME: ADÈLE LAST_NAME: MOLIÈRE

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

Example: Inserting Commas With MODIFY

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

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

A sample execution follows:

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

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

TRANSACTIONS: TOTAL = 1 ACCEPTED= 1 REJECTED= 0
SEGMENTS: INPUT = 1 UPDATED = 0 DELETED = 0

> The request processes as:

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

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

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

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

4. The request adds the transaction to the data source.
5. This request displays the semicolon converted to a comma:

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

The output is:

```
EMP_ID     LAST_NAME        FIRST_NAME  DEPARTMENT
------     ---------        ----------  ----------
224466880  BRADLEY, JR.     JOHN        MIS
```

**CTRFLD: Centering a Character String**

**How to:**

Center a Character String

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

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

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

**Syntax:**

**How to Center a Character String**

```
CTRFLD(source_string, length, output)
```

where:

- `source_string` 
  
  Alphanumeric 
  
  Is the character string enclosed in single quotation marks, or a field or variable that contains the character string.
**Example: Centering a Field**

CTRFLD centers `LAST_NAME` and stores the result in `CENTER_NAME`:

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

The output is:

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

**EDIT: Extracting or Adding Characters**

**How to:**

Extract or Add Characters

The EDIT function extracts characters from the source string and adds characters to the output string, according to the mask. It can extract a substring from different parts of the source string. It can also insert characters from the source string into an output string. For example, it can extract the first two characters and the last two characters of a string to form a single output string.
EDIT compares the characters in a mask to the characters in a source string. When it encounters a 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 321.

**Syntax:** *How to Extract or Add Characters*

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

where:

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

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

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

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

The output is:

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

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

**How to:**

Extract 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)

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

where:

- **source_string**
  Alphanumeric
  Is the source string from which to extract the token.

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

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

- **'delim'**
  Alphanumeric
  Is the delimiter in the source string enclosed in single quotation marks. If you specify more than one character, only the first character is used.

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

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

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

Note that the delimiter is not included in the extracted token.
**Example:** Extracting a Token

GETTOK extracts the last token from ADDRESS_LN3 and stores the result in LAST_TOKEN.

The delimiter is a space:

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

The output is:

<table>
<thead>
<tr>
<th>ADDRESS_LN3</th>
<th>LAST_TOKEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUTHERFORD NJ 07073</td>
<td>07073</td>
</tr>
<tr>
<td>NEW YORK NY 10039</td>
<td>10039</td>
</tr>
<tr>
<td>FREEPORT NY 11520</td>
<td>11520</td>
</tr>
<tr>
<td>NEW YORK NY 10001</td>
<td>10001</td>
</tr>
<tr>
<td>FREEPORT NY 11520</td>
<td>11520</td>
</tr>
<tr>
<td>ROSELAND NJ 07068</td>
<td>07068</td>
</tr>
<tr>
<td>JERSEY CITY NJ 07300</td>
<td>07300</td>
</tr>
<tr>
<td>FLUSHING NY 11354</td>
<td>11354</td>
</tr>
</tbody>
</table>

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

**How to:**

Convert a Character String to Mixed-Case

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

LCWORD skips numeric and special characters in the source string and continues to convert the following alphabetic characters. The result of LCWORD is a string in which the initial uppercase characters of all words are followed by lowercase characters.
**Syntax:** How to Convert a Character String to Mixed-Case

LCWORD(length, source_string, output)

where:

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

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

*output*  
Alphanumeric  
Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length must be greater than or equal to *length*.

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

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

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

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>MIXED_CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>Stevens</td>
</tr>
<tr>
<td>SMITH</td>
<td>Smith</td>
</tr>
<tr>
<td>BANNING</td>
<td>Banning</td>
</tr>
<tr>
<td>IRVING</td>
<td>Irving</td>
</tr>
<tr>
<td>ROMANS</td>
<td>Romans</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>Mcknight</td>
</tr>
</tbody>
</table>
LCWORD2: Converting a String to Mixed-Case

**How to:**
Convert a Character 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
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length must be greater than or equal to length.

**Example:**

Converting a Character String to Mixed-Case

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

```
DEFINE FILE EMPLOYEE
MYVAL1/A10='O'CONNOR'S';
LC2/A10 = LCWORD2(10, MYVAL1, 'A10');
END
TABLE FILE EMPLOYEE
SUM LAST_NAME NOPRINT MYVAL1 LC2
END
```
The output is:

```
MYVAL1      LC2
-----      ---
O'CONNOR'S  O'Connor's
```

**LCWORD3: Converting a String to Mixed-Case**

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

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
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length must be greater than or equal to length.
**Example:** Converting a Character String to Mixed-Case Using LCWORD3

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

```
DEFINE FILE EMPLOYEE
MYVAL1/A10='O'CONNOR'S';
MYVAL2/A10='o'connor's';
LC1/A10 = LCWORD3(10, MYVAL1, 'A10');
LC2/A10 = LCWORD3(10, MYVAL2, 'A10');
END FILE

TABLE FILE EMPLOYEE
SUM LAST_NAME NOPRINT MYVAL1 LC1 MYVAL2 LC2
END TABLE
```

On the output, the letter C after the first single quotation mark is in uppercase because it is not followed by a blank and is not the final letter in the input string. The letter s after the second single quotation mark is in lowercase because it is the last character in the input string:

<table>
<thead>
<tr>
<th>MYVAL1</th>
<th>LC1</th>
<th>MYVAL2</th>
<th>LC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>O'CONNOR'S</td>
<td>O'Connor's</td>
<td>o'connor's</td>
<td>O'Connor's</td>
</tr>
</tbody>
</table>

**LJUST: Left-Justifying a String**

**How to:**

Left-Justify a Character String

LJUST left-justifies a character string within a field. All leading spaces become trailing spaces.

LJUST will not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item.

There is a version of the LJUST function that is available only in the Maintain language. For information on this function, see *LJUST: Left-Justifying a Character String (Maintain)* on page 155.

**Syntax:**

How to Left-Justify a Character String

LJUST(length, source_string, output)

where:

- **length**
  - Integer

  Is the number of characters in source_string and output, or a field that contains the length.
**Example:**  **Left-Justifying a String**

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

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

TABLE FILE EMPLOYEE
PRINT LAST_NAME XNAME YNAME
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>XNAME</th>
<th>YNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>STEVENS</td>
<td>STEVENS</td>
</tr>
<tr>
<td>SMITH</td>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>JONES</td>
<td>JONES</td>
<td>JONES</td>
</tr>
<tr>
<td>SMITH</td>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>BANNING</td>
<td>BANNING</td>
<td>BANNING</td>
</tr>
<tr>
<td>IRVING</td>
<td>IRVING</td>
<td>IRVING</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ROMANS</td>
<td>ROMANS</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
<td>MCCOY</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>MCKNIGHT</td>
<td>MCKNIGHT</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
<td>CROSS</td>
</tr>
</tbody>
</table>
```
LOCASE: Converting Text to Lowercase

**How to:**
Convert Text to Lowercase

The LOCASE function converts alphanumeric text to lowercase.

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

**Syntax:**

**How to Convert Text to Lowercase**

LOCASE(length, source_string, output)

where:

*length*

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

*source_string*

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

*output*

- Alphanumeric
  - Is the name of the field in which to store the result, or the format of the output value enclosed in single quotation marks. The field name can be the same as *source_string*.

**Example:**

Converting a String to Lowercase

LOCASE converts the LAST_NAME field to lowercase and stores the result in LOWER_NAME:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
LOWER_NAME/A15 = LOCASE(15, LAST_NAME, LOWER_NAME);
WHERE DEPARTMENT EQ 'MIS';
END
```
### OVRLAY: Overlaying a Character String

#### How to:
Overlay a Character String

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

There is a version of the OVRLAY function that is available only in the Maintain language. For information on this function, see *OVRLAY: Overlaying a Character String (Maintain)* on page 162.

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

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

where:

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

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

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

---

The output is:

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

**OVRLAY: Overlaying a Character String**
sublen

Integer

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

position

Integer

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

output

Alphanumeric

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

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

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

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

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

The output is:

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

Using Functions
**Example:** Overlaying a Character in a String With MODIFY

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

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

The COMPUTE command invokes two functions:

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

The procedure processes as:

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

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

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

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

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

5. The procedure updates the first name in the data source.
6. When you exit the procedure with PF3, the transaction message indicates that one update occurred:

```
TRANSACTIONS:          TOTAL =  1  ACCEPTED=  1  REJECTED=  0
SEGMENTS:              INPUT =  0  UPDATED =  1  DELETED =  0
```

**PARAG: Dividing Text Into Smaller Lines**

**How to:**

Divide Text Into Smaller Lines

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

Once each token is marked off by the delimiter, you can use the function GETTOK to place the tokens into different fields (see [GETTOK: Extracting a Substring (Token)] on page 87). If PARAG does not find any spaces in the group it scans, it replaces the first character after the group with the delimiter. Therefore, make sure that any group of characters has at least one space. The number of characters scanned is provided as the maximum token size.

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

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

**Syntax:**

How to Divide Text Into Smaller Lines

```
PARAG(length, source_string, 'delimiter', max_token_size, output)
```
where:

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

**source_string**
Alphanumeric
Is a string to divide into tokens enclosed in single quotation marks, or a field or variable that contains the text.

**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.

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

**Example:**  **Dividing Text Into Smaller Lines**

PARAG divides ADDRESS_LN2 into smaller lines of not more than ten characters using a comma as the delimiter. It then stores the result in PARA_ADDR:

```plaintext
TABLE FILE EMPLOYEE
PRINT ADDRESS_LN2 AND COMPUTE
PARA_ADDR/A20 = PARAG(20, ADDRESS_LN2, ',', 10, PARA_ADDR);
BY LAST_NAME
WHERE TYPE EQ 'HSM';
END
```
The output is:

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

**PATTERN: Generating a Pattern From a String**

**How to:**

Generate a Pattern From an Input String

The PATTERN function examines a source string and produces a pattern that indicates the sequence of numbers, uppercase letters, and lowercase letters in the source string. This function is useful for examining data to make sure that it follows a standard pattern.

In the output pattern:

- Any character from the input that represents a single-byte digit becomes the character 9.
- Any character that represents an uppercase letter becomes A, and any character that represents a lowercase letter becomes a. For European NLS mode (Western Europe, Central Europe), A and a are extended to apply to accented alphabets.
- For Japanese, double-byte characters and Hankaku-katakana become C (uppercase). Note that double-byte includes Hiragana, Katakana, Kanji, full-width alphabets, full-width numbers, and full-width symbols. This means that all double-byte letters such as Chinese and Korean are also represented as C.
- Special characters remain unchanged.
- An unprintable character becomes the character X.
**Syntax:** How to Generate a Pattern From an Input String

PATTERN (length, source_string, output)

where:

*length*

Numeric

Is the length of *source_string*.

*source_string*

Alphanumeric

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

*output*

Alphanumeric

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

**Example:** Producing a Pattern From Alphanumeric Data

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

212-736-6250
212 736 4433
123-45-6789
800-969-INFO
10121-2898
10121
2 Penn Plaza
917-339-6380
917-339-4350
(212) 736-6250
(212) 736-4433
212-736-6250
212-736-6250
212-736-6250
(212) 736 5533
(212) 736 5533
(212) 736 5533
10121 E
800-969-INFO
The Master File is:

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

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

```plaintext
DYNAM ALLOC DD TESTFILE DA USER1.TESTFILE.FTM
DEFINE FILE TESTFILE
   PATTERN/A14 = PATTERN (14, TESTFLD, 'A14' ) ;
END

TABLE FILE TESTFILE
   SUM CNT.PATTERN AS 'COUNT' PCT.CNT.PATTERN AS 'PERCENT'
      BY PATTERN
PRINT TESTFLD
   BY PATTERN
ON TABLE COLUMN-TOTAL
END
```
Note that the next to last line produced a pattern from an input string that contained an unprintable character, so that character was changed to X. Otherwise, each numeric digit generated a 9 in the output string, each uppercase letter generated the character ‘A’, and each lowercase letter generated the character ‘a’. The output is:

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>COUNT</th>
<th>PERCENT</th>
<th>TESTFLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>(999) 999 9999</td>
<td>3</td>
<td>15.79</td>
<td>(212) 736 5533</td>
</tr>
<tr>
<td>(999) 999–9999</td>
<td>2</td>
<td>10.53</td>
<td>(212) 736–6250</td>
</tr>
<tr>
<td>9 Aaaa Aaaaa</td>
<td>1</td>
<td>5.26</td>
<td>2 Penn Plaza</td>
</tr>
<tr>
<td>999 999 9999</td>
<td>1</td>
<td>5.26</td>
<td>212 736 4433</td>
</tr>
<tr>
<td>999–99–9999</td>
<td>1</td>
<td>5.26</td>
<td>123–45–6789</td>
</tr>
<tr>
<td>999–999–AAAA</td>
<td>2</td>
<td>10.53</td>
<td>800–969–INFO</td>
</tr>
<tr>
<td>999–999–9999</td>
<td>6</td>
<td>31.58</td>
<td>212–736–6250</td>
</tr>
<tr>
<td>99999</td>
<td>1</td>
<td>5.26</td>
<td>10121</td>
</tr>
<tr>
<td>99999 X</td>
<td>1</td>
<td>5.26</td>
<td>10121 Æ</td>
</tr>
<tr>
<td>99999–9999</td>
<td>1</td>
<td>5.26</td>
<td>10121–2898</td>
</tr>
<tr>
<td>TOTAL</td>
<td>19</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

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

**How to:**

Find the Beginning of a Substring

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

There is a version of the POSIT function that is available only in the Maintain language. For information on this function, see *POSIT: Finding the Beginning of a Substring (Maintain)* on page 164.
**Syntax:**

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

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

where:

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

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

- **substring**
  - Alphanumeric
  - Is the substring whose position you want to find. This can be the substring enclosed in single quotation marks, or the field that contains the string.

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

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

POSIT determines the position of the first capital letter I in LAST_NAME and stores the result in I_IN_NAME:

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

The output is:

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

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

**How to:**
Reverse the Characters in a String

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

**Syntax:**

```
REVERSE(length, source_string, output)
```

where:

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

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

Alphanumeric

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

Example: Reversing the Characters in a String

In the following request against the EMPLOYEE data source, the REVERSE function is used to reverse the characters in the LAST_NAME field to produce the field named REVERSE_LAST. In this field, the trailing blanks from LAST_NAME have become leading blanks. The TRIM function is used to strip the leading blanks from REVERSE_LAST to produce the field named TRIM_REVERSE:

```
DEFINE FILE EMPLOYEE
REVERSE_LAST/A15 = REVERSE(15, LAST_NAME, REVERSE_LAST);
TRIM_REVERSE/A15 = TRIM('L', REVERSE_LAST, 15, ' ', 1, 'A15');
END

TABLE FILE EMPLOYEE
PRINT REVERSE_LAST TRIM_REVERSE
BY LAST_NAME
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>REVERSE_LAST</th>
<th>TRIM_REVERSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>GNINNAB</td>
<td>GNINNAB</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>DOOWKCALB</td>
<td>DOOWKCALB</td>
</tr>
<tr>
<td>CROSS</td>
<td>SSORC</td>
<td>SSORC</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>NAPSNEERG</td>
<td>NAPSNEERG</td>
</tr>
<tr>
<td>IRVING</td>
<td>GNVIRI</td>
<td>GNVIRI</td>
</tr>
<tr>
<td>JONES</td>
<td>SENOJ</td>
<td>SENOJ</td>
</tr>
<tr>
<td>MCCOY</td>
<td>YOCMM</td>
<td>YOCMM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>THGINKCM</td>
<td>THGINKCM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>SNAMOR</td>
<td>SNAMOR</td>
</tr>
<tr>
<td>SMITH</td>
<td>HTIMS</td>
<td>HTIMS</td>
</tr>
<tr>
<td>STEVENS</td>
<td>SNEVETS</td>
<td>SNEVETS</td>
</tr>
</tbody>
</table>
RJUST: Right-Justifying a Character String

How to:
Right-Justify 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.

RJUST does not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item. Also, if you use RJUST on a platform on which StyleSheets are turned on by default, issue SET STYLE=OFF before running the request.

There is a version of the RJUST function that is available only in the Maintain language. For information on this function, see RJUST: Right-Justifying a Character String (Maintain) on page 165.

Syntax: How to Right-Justify a Character String

RJUST(length, source_string, output)

where:

length
Integer

Is the number of characters in source_string and output, or a field that contains the length. Their lengths must be the same to avoid justification problems.

source_string
Alphanumeric

Is the character string to right justify, or a field or variable that contains the character string enclosed in single quotation marks.

output
Alphanumeric

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

RJUST right-justifies the LAST_NAME field and stores the result in RIGHT_NAME:

```plaintext
SET STYLE=OFF
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
RIGHT_NAME/A15 = RJUST(15, LAST_NAME, RIGHT_NAME); WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

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

**SOUNDEX: Comparing Character Strings Phonetically**

**How to:**

Compare Character Strings Phonetically

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

To conduct a phonetic search, do the following:

1. Use SOUNDEX to translate data values from the field you are searching for to the phonetic codes.
2. Use SOUNDEX to translate your best guess target string to a phonetic code. Remember that the spelling of your target string need be only approximate; however, the first letter must be correct.
3. Use WHERE or IF criteria to compare the temporary fields created in Step 1 to the temporary field created in Step 2.
**Syntax:**  
How to Compare Character Strings Phonetically

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

where:

- **length**
  
  Alphanumeric
  
  Is the number of characters in `source_string`, or a field that contains the length. It can be a number enclosed in single quotation marks, or a field containing the number. The number must be from 01 to 99, expressed with two digits (for example '01'); a number larger than 99 causes the function to return asterisks (*) as output.

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

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

**Example:**  
Comparing Character Strings Phonetically

The following request creates three fields:

- PHON_NAME contains the phonetic code of employee last names.
- PHON_COY contains the phonetic code of your guess, MICOY.
- PHON_MATCH contains YES if the phonetic codes match, NO if they do not.

The WHERE criteria selects the last name that matches your best guess.

```
DEFINE FILE EMPLOYEE
PHON_NAME/A4 = SOUNDEX('15', LAST_NAME, PHON_NAME);
PHON_COY/A4 WITH LAST_NAME = SOUNDEX('15', 'MICOY', PHON_COY);
PHON_MATCH/A3 = IF PHON_NAME IS PHON_COY THEN 'YES' ELSE 'NO';
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME
IF PHON_MATCH IS 'YES'
END
```
The output is:

LAST_NAME
---------
MCCOY

**SPELLNM: Spelling Out a Dollar Amount**

**How to:**
Spell Out a Dollar Amount

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

**Syntax:**

`SPELLNM(outlength, number, output)`

where:

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

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

<table>
<thead>
<tr>
<th>If number is less than...</th>
<th>...outlength should be</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>37</td>
</tr>
<tr>
<td>$100</td>
<td>45</td>
</tr>
<tr>
<td>$1,000</td>
<td>59</td>
</tr>
<tr>
<td>$10,000</td>
<td>74</td>
</tr>
<tr>
<td>$100,000</td>
<td>82</td>
</tr>
<tr>
<td>$1,000,000</td>
<td>96</td>
</tr>
</tbody>
</table>
number

Alphanumeric or Numeric (9.2)

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

output

Alphanumeric

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

Example:  **Spelling Out a Dollar Amount**

SPELLNM spells out the values in CURR_SAL and stores the result in AMT_IN_WORDS:

```plaintext
TABLE FILE EMPLOYEE
PRINT CURR_SAL AND COMPUTE
AMT_IN_WORDS/A82 = SPELLNM(82, CURR_SAL, AMT_IN_WORDS);
WHERE DEPARTMENT EQ 'MIS'
END
```

The output is:

<table>
<thead>
<tr>
<th>CURR_SAL</th>
<th>AMT_IN_WORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$13,200.00</td>
<td>THIRTEEN THOUSAND TWO HUNDRED DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$18,480.00</td>
<td>EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$18,480.00</td>
<td>EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$21,780.00</td>
<td>TWENTY-ONE THOUSAND SEVEN HUNDRED EIGHTY DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$9,000.00</td>
<td>NINE THOUSAND DOLLARS AND NO CENTS</td>
</tr>
<tr>
<td>$27,062.00</td>
<td>TWENTY-SEVEN THOUSAND SIXTY-TWO DOLLARS AND NO CENTS</td>
</tr>
</tbody>
</table>

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

**How to:**

Reduce Multiple Spaces to a Single Space

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

Syntax:

SQUEEZ(length, source_string, output)

where:

length

Integer

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

source_string

Alphanumeric

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

output

Alphanumeric

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

Example: Reducing Multiple Spaces to a Single Space

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

DEFINE FILE EMPLOYEE
NAME/A30 = FIRST_NAME | LAST_NAME;
END
TABLE FILE EMPLOYEE
PRINT NAME AND COMPUTE
SQNAME/A30 = SQUEEZ(30, NAME, 'A30');
WHERE DEPARTMENT EQ 'MIS';
END

The output is:

<table>
<thead>
<tr>
<th>NAME</th>
<th>SQNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARY</td>
<td>SMITH</td>
</tr>
<tr>
<td>DIANE</td>
<td>JONES</td>
</tr>
<tr>
<td>JOHN</td>
<td>MCCOY</td>
</tr>
<tr>
<td>ROSEMARIE</td>
<td>BLACKWOOD</td>
</tr>
<tr>
<td>MARY</td>
<td>GREENSPAN</td>
</tr>
<tr>
<td>BARBARA</td>
<td>CROSS</td>
</tr>
</tbody>
</table>

MARY       SMITH          MARY SMITH
DIANE      JONES          DIANE JONES
JOHN       MCCOY          JOHN MCCOY
ROSEMARIE  BLACKWOOD      ROSEMARIE BLACKWOOD
MARY       GREENSPAN      MARY GREENSPAN
BARBARA    CROSS          BARBARA CROSS
STRIP: Removing a Character From a String

How to:
Remove a Character From a String

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

Syntax:

How to Remove a Character From a String

STRIP(length, source_string, char, output)

where:

*length*

Integer

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

*source_string*

Alphanumeric

Is the string from which the character will be removed, or a field containing the string.

*char*

Alphanumeric

Is the character to be removed from the string. This can be an alphanumeric literal enclosed in single quotation marks, or a field that contains the character. If more than one character is provided, the left-most character will be used as the strip character.

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

*output*

Alphanumeric

Is the field that contains the result, or the format of the output value enclosed in single quotation marks.
Example: Removing Occurrences of a Character From a String

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

```
TABLE FILE MOVIES
PRINT DIRECTOR AND COMPUTE
SDIR/A17 = STRIP(17, DIRECTOR, '.', 'A17');
WHERE CATEGORY EQ 'COMEDY'
END
```

The output is:

<table>
<thead>
<tr>
<th>DIRECTORS</th>
<th>SDIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEMECKIS R.</td>
<td>ZEMECKIS R</td>
</tr>
<tr>
<td>ABRAHAMS J.</td>
<td>ABRAHAMS J</td>
</tr>
<tr>
<td>ALLEN W.</td>
<td>ALLEN W</td>
</tr>
<tr>
<td>HALLSTROM L.</td>
<td>HALLSTROM L</td>
</tr>
<tr>
<td>MARSHALL P.</td>
<td>MARSHALL P</td>
</tr>
<tr>
<td>BROOKS J.L.</td>
<td>BROOKS JL</td>
</tr>
</tbody>
</table>

Example: Removing Single Quotation Marks From a String

STRIP removes all occurrences of a single quotation mark ('') from the TITLE field and stores the result in a field with the format A39:

```
TABLE FILE MOVIES
PRINT TITLE AND COMPUTE
STITLE/A39 = STRIP(39, TITLE, '''', 'A39');
WHERE TITLE CONTAINS '''
END
```

The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>STITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BABETTE'S FEAST</td>
<td>BABETTES FEAST</td>
</tr>
<tr>
<td>JANE FONDA'S COMPLETE WORKOUT</td>
<td>JANE FONDAS COMPLETE WORKOUT</td>
</tr>
<tr>
<td>JANE FONDA'S NEW WORKOUT</td>
<td>JANE FONDAS NEW WORKOUT</td>
</tr>
<tr>
<td>MICKEY MANTLE'S BASEBALLTIPS</td>
<td>MICKEY MANTLES BASEBALL TIPS</td>
</tr>
</tbody>
</table>
**Example:** Removing Commas From a String (Maintain)

STRIP removes all occurrences of a comma from the TITLE field:

```
MAINTAIN FILE MOVIES
FOR 10 NEXT MOVIECODE INTO MOVSTK
  WHERE TITLE CONTAINS ",";
COMPUTE I/I2=1;
REPEAT MOVSTK.FOCINDEX
  TYPE "TITLE IS: <MOVSTK(I).TITLE"
COMPUTE NOCOMMA/A39=STRIP(39,MOVSTK().TITLE, ",",NOCOMMA);
  TYPE "NEW TITLE IS: <NOCOMMA"
COMPUTE I=I+1
ENDREPEAT
END
```

The output is:

```
TITLE IS: SMURFS, THE
NEW TITLE IS: SMURFS THE
```

**STRREP: Replacing Character Strings**

**How to:**
Replace Character Strings

**Reference:**
Usage Notes for STRREP Function

The STRREP replaces all instances of a specified string within a source string. It also supports replacement by null strings.

**Syntax:** How to Replace Character Strings

```
STRREP (inlength, instring, searchlength, searchString, replength, repstring, outlength, output)
```

where:

- `inlength`
  - Numeric
  - Is the number of characters in the source string.

- `instring`
  - Alphanumeric
  - Is the source string.
**searchlength**
   Numeric
   Is the number of characters in the (shorter length) string to be replaced.

**searchstring**
   Alphanumeric
   Is the character string to be replaced.

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

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

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

**output**
   Alphanumeric
   Is the resulting output string after all replacements and padding.

**Reference:** Usage Notes for STRREP Function
The maximum string length is 4095.
Example:  Replacing Commas and Dollar Signs

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

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

The output is:

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

SUBSTR: Extracting a Substring

How to:  Extract a Substring

The SUBSTR function extracts a substring based on where it begins and its length in the source string. SUBSTR can vary the position of the substring depending on the values of other fields.

There is a version of the SUBSTR function that is available only in the Maintain language. For information on this function, see SUBSTR: Extracting a Substring (Maintain) on page 173.
**Syntax:**

How to Extract a Substring

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

where:

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

- **source_string**
  - Alphanumeric
  - Is the string from which to extract a substring enclosed in single quotation marks, or the field containing the parent string.

- **start**
  - Integer
  - Is the starting position of the substring in the source string. If `start` is less than one or greater than `length`, the function returns spaces.

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

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

- **output**
  - Alphanumeric
  - Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.
**Example: Extracting a String**

POSIT determines the position of the first letter I in LAST_NAME and stores the result in I_IN_NAME. SUBSTR then extracts three characters beginning with the letter I from LAST_NAME, and stores the results in I_SUBSTR.

```
TABLE FILE EMPLOYEE
PRINT
COMPUTE
  I_IN_NAME/I2 = POSIT(LAST_NAME, 15, 'I', 1, 'I2'); AND
COMPUTE
  I_SUBSTR/A3 =
    SUBSTR(15, LAST_NAME, I_IN_NAME, I_IN_NAME+2, 3, I_SUBSTR);
BY LAST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION'
END
```

The output is:

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

Since Romans and Stevens have no I in their names, SUBSTR extracts a blank string.
TRIM: Removing Leading and Trailing Occurrences

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

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

There is a version of the TRIM function that is available only in the Maintain language. For information on this function, see *TRIM: Removing Trailing Occurrences (Maintain)* on page 175.

**Syntax:**

```plaintext
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 enclosed in single quotation marks, or the field containing the string.

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

- `pattern`
  - Alphanumeric
  - Is the character string pattern to remove enclosed in single quotation marks.

- `sublength`
  - Integer
  - Is the number of characters in the pattern.
**Example: ** Removing Leading Occurrences

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

```
DEFINE FILE MOVIES
SHORT/A19 = SUBSTR(19, TITLE, 1, 19, 19, SHORT);
END
```

```
TABLE FILE MOVIES
PRINT  DIRECTOR AND
COMPUTE
   TRIMDIR/A17 = TRIM('L', DIRECTOR, 17, 'BR', 2, 'A17');
   WHERE DIRECTOR CONTAINS 'BR'
END
```

The output is:

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

**Example: ** Removing Trailing Occurrences

TRIM removes trailing occurrences of the characters ER from the TITLE. In order to remove trailing non-blank characters, trailing spaces must be removed first. The TITLE field has trailing spaces. Therefore, TRIM does not remove the characters ER when creating field TRIMT. The SHORT field does not have trailing spaces. Therefore, TRIM removes the trailing ER characters when creating field TRIMS:

```
DEFINE FILE MOVIES
SHORT/A19 = SUBSTR(19, TITLE, 1, 19, 19, SHORT);
END
```

```
TABLE FILE MOVIES
PRINT  TITLE  IN 1  AS 'TITLE: '  
      SHORT  IN 40 AS 'SHORT: ' OVER
COMPUTE
   TRIMT/A39 = TRIM('T', TITLE, 39, 'ER', 2, 'A39'); IN 1 AS 'TRIMT: '  
   WHERE TITLE LIKE '%ER'
COMPUTE
   TRIMS/A19 = TRIM('T', SHORT, 19, 'ER', 2, 'A19'); IN 40 AS 'TRIMS: '  
   WHERE TITLE LIKE '%ER'
END
```
The output is:

| TITLE:         | LEARN TO SKI BETTER                     | SHORT:      | LEARN TO SKI BETTER |
| TRIMT:         | LEARN TO SKI BETTER                     | TRIMS:      | LEARN TO SKI BETT   |
| TITLE:         | FANNY AND ALEXANDER                     | SHORT:      | FANNY AND ALEXANDER |
| TRIMT:         | FANNY AND ALEXANDER                     | TRIMS:      | FANNY AND ALEXANDR  |

**UPCASE: Converting Text to Uppercase**

### How to:

Convert Text to Uppercase

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

In FIDEL, CRTFORM LOWER retains the case of entries exactly as they were typed. Use **UPCASE** to convert entries for particular fields to uppercase.

There is a version of the **UPCASE** function that is available only in the Maintain language. For information on this function, see **UPCASE: Converting Text to Uppercase (Maintain)** on page 176.

### Syntax: How to Convert Text to Uppercase

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

where:

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

- **input**
  
  Alphanumeric
  
  Is the string to convert enclosed in single quotation marks, or the field containing the character string.

- **output**
  
  Alphanumeric of type \( \text{A}\text{n} \) or \( \text{A}\text{n} \)
  
  Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.
**Example: Converting a Mixed-Case String to Uppercase**

UPCASE converts the LAST_NAME_MIXED field to uppercase:

```plaintext
DEFINE FILE EMPLOYEE
LAST_NAME_MIXED/A15=IF DEPARTMENT EQ 'MIS' THEN LAST_NAME ELSE LCWORD(15, LAST_NAME, 'A15');
LAST_NAME_UPPER/A15=UPCASE(15, LAST_NAME_MIXED, 'A15');
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME_MIXED AND FIRST_NAME BY LAST_NAME_UPPER
WHERE CURR_JOBCODE EQ 'B02' OR 'A17' OR 'B04';
END

Now, when you execute the request, the names are sorted correctly.

The output is:

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

If you do not want to see the field with all uppercase values, you can NOPRINT it.

**Example: Converting a Lowercase Field to Uppercase With MODIFY**

Suppose your company decides to store employee names in mixed case and the department assignments in uppercase.

To enter records for new employees, execute this MODIFY procedure:

```plaintext
MODIFY FILE EMPLOYEE
CRTFORM LOWER
"ENTER EMPLOYEE'S ID : <EMP_ID"
"ENTER LAST_NAME: <LAST_NAME FIRST_NAME: <FIRST_NAME"
"TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET"
""
"ENTER DEPARTMENT ASSIGNMENT: <DEPARTMENT"
MATCH EMP_ID
ON MATCH REJECT
ON NOMATCH COMPUTE
   DEPARTMENT = UPCASE(10, DEPARTMENT, 'A10');
ON NOMATCH INCLUDE
   ON NOMATCH TYPE "DEPARTMENT VALUE CHANGED TO UPPERCASE: <DEPARTMENT"
DATA
END
```

124 Information Builders
The procedure processes as:

1. The procedure prompts you for an employee ID, last name, first name, and department on a CRTFORM screen. The CRTFORM LOWER option retains the case of entries exactly as typed.

2. You type the following data and press Enter:

   ENTER EMPLOYEE'S ID: 444555666
   ENTER LAST_NAME: Cutter                    FIRST_NAME: Alan
   TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET
   ENTER DEPARTMENT ASSIGNMENT: sales

3. The procedure searches the data source for the ID 444555666. If it does not find the ID, it continues processing the transaction.

4. UPCASE converts the DEPARTMENT entry sales to SALES:

   ENTER EMPLOYEE'S ID:
   ENTER LAST_NAME:                     FIRST_NAME:
   TYPE THE NAME EXACTLY AS YOU SEE IT ON THE SHEET
   ENTER DEPARTMENT ASSIGNMENT:
   DEPARTMENT VALUE CHANGED TO UPPERCASE: SALES

5. The procedure adds the transaction to the data source.

6. When you exit the procedure with PF3, the transaction message indicates the number of transactions accepted or rejected:

   TRANSACTIONS: TOTAL = 1 ACCEPTED= 1 REJECTED= 0
   SEGMENTS: INPUT = 1 UPDATED = 0 DELETED = 0
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.

**Topics:**
- 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
For relational data sources, \texttt{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, \texttt{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, \texttt{AnV} format is not recommended for use with non-relational data sources.

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

The functions described in this topic are designed to work specifically with the \texttt{AnV} data type parameters.

**Reference**: Usage Notes for Using an AnV Field in a Function

The following affect the use of an AnV field in a function:

- When using an \texttt{AnV} argument in a function, the input parameter is treated as an \texttt{An} parameter and is padded with blanks to its declared size \((n)\). If the last parameter specifies an \texttt{AnV} format, the function result is converted to type \texttt{AnV} with actual length set equal to its size.

- Many functions require both an alphanumeric string and its length as input arguments. If the supplied string is stored in an \texttt{AnV} field, you still must supply a length argument to satisfy the requirements of the function. However, the length that will be used in the function's calculations is the actual length stored as the first two bytes of the \texttt{AnV} field.

- In general, any input argument can be a field or a literal. In most cases, numeric input arguments are supplied to these functions as literals, and there is no reason not to supply an integer value. However, if the value is not an integer, it is truncated to an integer value regardless of whether it was supplied as a field or a literal.
LENV: Returning the Length of an Alphanumeric Field

**How to:**
Find 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

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

where:

- \text{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.

- \text{output}
  - Integer
  - Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.

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

TRIMV creates an AnV field named TITLEV by removing trailing blanks from the TITLE value. Then LENV returns the actual length of each instance of TITLEV to the ALEN field:

\[
\begin{align*}
\text{TABLE FILE MOVIES} \\
\text{PRINT} \\
\text{COMPUTE TITLEV/39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);} \\
\text{ALEN/I2 = LENV(TITLEV, ALEN);} \\
\text{BY CATEGORY NOPRINT} \\
\text{WHERE CATEGORY EQ 'CHILDREN'} \\
\text{END}
\end{align*}
\]
LOCASV: Creating a Variable Length Lowercase String

The output is:

<table>
<thead>
<tr>
<th>TITLEV</th>
<th>ALEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>11</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>15</td>
</tr>
<tr>
<td>SCOOBY-DOO-A DOG IN THE RUFF</td>
<td>28</td>
</tr>
<tr>
<td>ALICE IN WONDERLAND</td>
<td>19</td>
</tr>
<tr>
<td>SESAME STREET-BEDTIME STORIES AND SONGS</td>
<td>39</td>
</tr>
<tr>
<td>ROMPER ROOM-ASK MISS MOLLY</td>
<td>26</td>
</tr>
<tr>
<td>SLEEPING BEAUTY</td>
<td>15</td>
</tr>
<tr>
<td>BAMBI</td>
<td>5</td>
</tr>
</tbody>
</table>

How to:
Create a Variable Length Lowercase String

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

Syntax: How to Create a Variable Length Lowercase String

LOCASV(upper_limit, source_string, output)

Where:

upper_limit
Integer
Is the limit for the length of the source string.

source_string
Alphanumeric of type An or AnV
Is the string to be converted to lowercase in single quotation marks, or a field or variable that contains the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If upper_limit is smaller than the actual length, the source string is truncated to this upper limit.

output
Alphanumeric of type An or AnV
Is the name of the field in which to store the result, or the format of the output value enclosed in single quotation marks. This value can be for a field that is AnV or An format.

If the output format is AnV, the actual length returned is equal to the smaller of the source string length and the upper limit.
**Example:** Creating a Variable Length Lowercase String

In this example, LOCASV converts the LAST_NAME field to lowercase and specifies a length limit of five characters. The results are stored in the LOWCV_NAME field:

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND COMPUTE
LOWCV_NAME/A15V = LOCASV(5, LAST_NAME, LOWCV_NAME);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>LOWCV_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>smith</td>
</tr>
<tr>
<td>JONES</td>
<td>jones</td>
</tr>
<tr>
<td>MCCOY</td>
<td>mccooy</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>black</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>green</td>
</tr>
<tr>
<td>CROSS</td>
<td>cross</td>
</tr>
</tbody>
</table>

**POSITV: Finding the Beginning of a Variable Length Substring**

**How to:**

Find 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. It can be the string enclosed in single quotation marks, or a field or variable that contains the source string. If it is a field of AnV format, its length is taken from the length bytes stored in the field. If upper_limit is smaller than the actual length, the source string is truncated to this upper limit.
**upper_limit**

Integer

Is a limit for the length of the source string.

**substring**

Alphanumeric of type An or AnV

Is the substring whose position you want to find. This can be the substring enclosed in single quotation marks, or the field that contains the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length bytes stored in the field. If sub_limit is smaller than the actual length, the source string is truncated to this limit.

**sub_limit**

Integer

Is limit for the length of the substring.

**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:**  **Finding the Starting Position of a Variable Length Pattern**

POSITV finds the starting position of a trailing definite or indefinite article in a movie title (such as ", THE" in SMURFS, THE). First TRIMV removes the trailing blanks from the title so that the article will be the trailing pattern:

```
DEFINE FILE MOVIES
  TITLEV/A39V = TRIMV('T',TITLE, 39,' ', 1, TITLEV);
  PSTART/I4 = POSITV(TITLEV,LENV(TITLEV,'I4'), ',', 1,'I4');
  PLEN/I4 = IF PSTART NE 0 THEN LENV(TITLEV,'I4') - PSTART +1 ELSE 0;
END
```

```
TABLE FILE MOVIES
  PRINT TITLE
    PSTART AS 'Pattern,Start' IN 25
    PLEN AS 'Pattern,Length'
  BY CATEGORY  NOPRINT
  WHERE PLEN NE 0
END
```
The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>Pattern Start</th>
<th>Pattern Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMURFS, THE</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>MALTESE FALCON, THE</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>PHILADELPHIA STORY, THE</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>TIN DRUM, THE</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>FAMILY, THE</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>MORNING AFTER, THE</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>BIRDS, THE</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>BOY AND HIS DOG, A</td>
<td>16</td>
<td>3</td>
</tr>
</tbody>
</table>

**SUBSTV: Extracting a Variable Length Substring**

**How to:**
Extract a Variable Length Substring

The SUBSTV function extracts a substring from a string and is similar to SUBSTR. However, the end position for the string is calculated from the starting position and the substring length. Therefore, it has fewer parameters than SUBSTR. Also, the actual length of the output field, if it is an 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. It can be the string enclosed in single quotation marks, or the field containing the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length bytes stored in the field. If `upper_limit` is smaller than the actual length, the source string is truncated to the upper limit. The final length value determined by this comparison is referred to as `p_length` (see the description of the `output` parameter for related information).
**start**

Integer

Is the starting position of the substring in the source string. The starting position can exceed the source string length, which results in spaces being returned.

**sub_limit**

Integer

Is the length, in characters, of the substring (normally end - start + 1). The end position of the substring is end = start + sublength - 1. Note that the ending position can exceed the input string length depending on the provided values for start and sub_limit.

**output**

Alphanumeric of type An or AnV

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

If the format of output is AnV, the actual length, outlen, is computed as follows from the values for end, start, and p_length (see the source_string parameter for related information):

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

**Example: Extracting a Variable Length Substring**

The following request extracts a trailing definite or indefinite article from a movie title (such as " , THE" in "SMURFS, THE"). First it trims the trailing blanks so that the article is the trailing pattern. Next it finds the starting position and length of the pattern. Then SUBSTV extracts the pattern and TRIMV trims the pattern from the title:

```
DEFINE FILE MOVIES
  TITLEV/A39V = TRIMV('T',TITLE, 39,' ', 1, TITLEV);
  PSTART/I4 = POSITV(TITLEV,LENV(TITLEV,'I4'), ',', 1,'I4');
  PLEN/I4 = IF PSTART NE 0 THEN LENV(TITLEV,'I4') - PSTART +1   
            ELSE 0;
  PATTERN/A20V= SUBSTV(39, TITLEV, PSTART, PLEN, PATTERN);
  NEWTIT/A39V = TRIMV('T',TITLE,39,PATTERN,LENV(PATTERN,'I4'), NEWTIT);
END
TABLE FILE MOVIES
  PRINT TITLE
    PSTART AS 'Pattern,Start' IN 25
    PLEN AS 'Pattern,Length'
  NEWTIT AS 'Trimmed,Title' IN 55
  BY CATEGORY  NOPRINT
WHERE PLEN NE 0
END
```
The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>Pattern</th>
<th>Pattern</th>
<th>Trimmed</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>Start</td>
<td>Length</td>
<td>Title</td>
</tr>
<tr>
<td>SMURFS, THE</td>
<td>7</td>
<td>5</td>
<td>SMURFS</td>
</tr>
<tr>
<td>SHAGGY DOG, THE</td>
<td>11</td>
<td>5</td>
<td>SHAGGY DOG</td>
</tr>
<tr>
<td>MALTESE FALCON, THE</td>
<td>15</td>
<td>5</td>
<td>MALTESE FALCON</td>
</tr>
<tr>
<td>PHILADELPHIA STORY, THE</td>
<td>19</td>
<td>5</td>
<td>PHILADELPHIA STORY</td>
</tr>
<tr>
<td>TIN DRUM, THE</td>
<td>9</td>
<td>5</td>
<td>TIN DRUM</td>
</tr>
<tr>
<td>FAMILY, THE</td>
<td>7</td>
<td>5</td>
<td>FAMILY</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>12</td>
<td>3</td>
<td>CHORUS LINE</td>
</tr>
<tr>
<td>MORNING AFTER, THE</td>
<td>14</td>
<td>5</td>
<td>MORNING AFTER</td>
</tr>
<tr>
<td>BIRDS, THE</td>
<td>6</td>
<td>5</td>
<td>BIRDS</td>
</tr>
<tr>
<td>BOY AND HIS DOG, A</td>
<td>16</td>
<td>3</td>
<td>BOY AND HIS DOG</td>
</tr>
</tbody>
</table>

**TRIMV: Removing Characters From a String**

**How to:**

Remove 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. It can be the string enclosed in single quotation marks, or the field containing the string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If upper_limit is smaller than the actual length, the source string is truncated to this upper limit.

slength_limit
Integer
Is limit for the length of the source string.

pattern
Alphanumeric of type An or AnV
Is the pattern to remove enclosed in single quotation marks. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If pattern_limit is smaller than the actual length, the pattern is truncated to this limit.

plength_limit
Integer
Is the limit for the length of the pattern.

output
Alphanumeric of type An or AnV
Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks. The field can be in AnV or An format.
If the output format is AnV, the length is set to the number of characters left after trimming.

**Example:** Creating an AnV Field by Removing Trailing Blanks
TRIMV creates an AnV field named TITLEV by removing trailing blanks from the TITLE value:

```
TABLE FILE MOVIES
PRINT DIRECTOR
COMPUTE TITLEV/A39V = TRIMV('T', TITLE, 39, ' ', 1, TITLEV);
BY CATEGORY
END
```
Here are the first 10 lines of the output:

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>DIRECTOR</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>SPIELBERG S.</td>
<td>JAWS</td>
</tr>
<tr>
<td></td>
<td>VERHOVEN P.</td>
<td>ROBOCOP</td>
</tr>
<tr>
<td></td>
<td>VERHOVEN P.</td>
<td>TOTAL RECALL</td>
</tr>
<tr>
<td></td>
<td>SCOTT T.</td>
<td>TOP GUN</td>
</tr>
<tr>
<td></td>
<td>MCDONALD P.</td>
<td>RAMBO III</td>
</tr>
<tr>
<td>CHILDREN</td>
<td></td>
<td>SMURFS, THE</td>
</tr>
<tr>
<td></td>
<td>BARTON C.</td>
<td>SHAGGY DOG, THE</td>
</tr>
<tr>
<td></td>
<td>GEROMINI</td>
<td>ALICE IN WONDERLAND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SESAME STREET-BEDTIME STORIES AND SONGS</td>
</tr>
</tbody>
</table>

UPCASV: Creating a Variable Length Uppercase String

**How to:**
Create 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**

\[
\text{UPCASV}(\text{upper-limit}, \text{source-string}, \text{output})
\]

where:

- **upper-limit**
  - Integer
  - Is the limit for the length of the source string. It can be a positive constant or a field whose integer portion represents the upper limit.

- **source-string**
  - Alphanumeric of type An or AnV
  - is the string to convert to uppercase. It can be the character string enclosed in single quotation marks, or the field containing the character string. If it is a field, it can have An or AnV format. If it is a field of type AnV, its length is taken from the length in bytes stored in the field. If **upper-limit** is smaller than the actual length, the source string is truncated to the upper limit.
output

Alphanumeric of type An or AnV

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks. This can be a field with AnV or An format.

If the output format is AnV, the length returned is equal to the smaller of the source string length and upper_limit.

**Example:** Creating a Variable Length Uppercase String

Suppose you are sorting on a field that contains both uppercase and mixed-case values. The following request defines a field called LAST_NAME_MIXED that contains both uppercase and mixed-case values:

```
DEFINE FILE EMPLOYEE
LAST_NAME_MIXED/A15=IF DEPARTMENT EQ 'MIS' THEN LAST_NAME ELSE LCWORD(15, LAST_NAME, 'A15');
LAST_NAME_UPCASV/A15V=UPCASV(5, LAST_NAME_MIXED, 'A15');
END
```

Suppose you execute a request that sorts by this field:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME_MIXED AND FIRST_NAME BY LAST_NAME_UPCASV
WHERE CURR_JOBCODE EQ 'B02' OR 'A17' OR 'B04';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME_UPCASV</th>
<th>LAST_NAME_MIXED</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNI</td>
<td>Banning</td>
<td>JOHN</td>
</tr>
<tr>
<td>BLACK</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>CROSS</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>MCCOY</td>
<td>MCCOY</td>
<td>JOHN</td>
</tr>
<tr>
<td>MCKNI</td>
<td>Mcknight</td>
<td>ROGER</td>
</tr>
<tr>
<td>ROMAN</td>
<td>Romans</td>
<td>ANTHONY</td>
</tr>
</tbody>
</table>
The functions in this topic manipulate strings of DBCS and SBCS characters when your configuration uses a DBCS code page.

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

**How to:**

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

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

**Syntax:**

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

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

where:

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

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

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

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

- **output**
  - Alphanumeric
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
Using DCTRAN to Translate Double-Byte Characters

Example: Using DCTRAN to Translate Double-Byte Characters

In the following:

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

For \(A\text{¥}A本B語\), the result is \(A\text{¥}A本B語\).

DEDIT: Extracting or Adding Characters

**How to:**

Extract or Add DBCS or SBCS Characters

If your configuration uses a DBCS code page, you can use the DEDIT function to extract characters 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**

\[
\text{DEDIT}(\text{inlength}, \text{source_string}, \text{mask_length}, \text{mask}, \text{output})
\]

where:

\[
\text{inlength}
\]

Integer

Is the number of bytes in \text{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.

\[
\text{source_string}
\]

Alphanumeric

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

\[
\text{mask_length}
\]

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

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

Example: Adding and Extracting DBCS Characters

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

DEDIT( 15, 'あいうえお', 16, '99$9$9$9$9$-かきくけこ', 'A30')

The result is あいうえお-かきくけこ.

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

DEDIT( 15, 'あいうえお', 16, '99$9$9$9$9-$ABCDE', 'A20')

The result is あいうえお-ABCDE.

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

How to:

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

The DSTRIP function removes all occurrences of a specific single-byte or double-byte character from a string. The resulting character string has the same length as the original string but is padded on the right with spaces.
**Syntax:** How to Remove a Single-Byte or Double-Byte Character From a String

DSTRIP(length, source_string, char, output)

where:

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

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

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

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

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

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

In the following:

DSTRIP(9, 'Ａ日Ａ本Ｂ語', '日', A9)

For Ａ日Ａ本Ｂ語, the result is AA本B語.

**DSUBSTR: Extracting a Substring**

**How to:** Extract a Substring

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

**How to Extract a Substring**

DSUBSTR(inlength, source_string, start, end, sublength, output)

where:

*inlength*

Integer

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

*source_string*

Alphanumeric

Is the string from which the substring will be extracted enclosed in single quotation marks, or the field containing the parent string.

*start*

Integer

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

*end*

Integer

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

*sublength*

Integer

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

*output*

Alphanumeric

Is the field to which the result is returned, or the format of the output value enclosed in single quotation marks.
**Example: Extracting a Substring**

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

```
DSTR(15, 'あいうえお', 4, 6, 'A10')
The result is いう。
```

**JPTTRANS: Converting Japanese Specific Characters**

**How to:**
Convert Japanese Specific Characters

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

The JPTTRANS function converts Japanese specific characters.

**Syntax:** How to Convert Japanese Specific Characters

JPTTRANS ('type_of_conversion', length, source_string, 'output_format')

where:

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

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'UPCASE'</td>
<td>Converts Zenkaku (Fullwidth) alphabets to Zenkaku uppercase.</td>
</tr>
<tr>
<td>'LOCASE'</td>
<td>Converts Zenkaku alphabets to Zenkaku lowercase.</td>
</tr>
<tr>
<td>'HNZNALPHA'</td>
<td>Converts alphanumerics from Hankaku (Halfwidth) to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNSIGN'</td>
<td>Converts ASCII symbols from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNKANA'</td>
<td>Converts Katakana from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'HNZNSPACE'</td>
<td>Converts space (blank) from Hankaku to Zenkaku.</td>
</tr>
<tr>
<td>'ZNHNALPHA'</td>
<td>Converts alphanumerics from Zenkaku to Hankaku.</td>
</tr>
</tbody>
</table>
## JPTRANS: Converting Japanese Specific Characters

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ZNHNSIGN'</td>
<td>Converts ASCII symbols from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZNHNKANA'</td>
<td>Converts Katakana from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'ZNHNSPACE'</td>
<td>Converts space from Zenkaku to Hankaku.</td>
</tr>
<tr>
<td>'HIRAKATA'</td>
<td>Converts Hiragana to Zenkaku Katakana.</td>
</tr>
<tr>
<td>'KATAHIRA'</td>
<td>Converts Zenkaku Katakana to Hiragana.</td>
</tr>
<tr>
<td>'930TO939'</td>
<td>Converts codepage from 930 to 939.</td>
</tr>
<tr>
<td>'939TO930'</td>
<td>Converts codepage from 939 to 930.</td>
</tr>
</tbody>
</table>

### length

Integer

Is the number of characters in the source_string.

### source_string

Alphanumeric

Is the string to convert.

### output_format

Alphanumeric

Is the name of the field that contains the output, or the format enclosed in single quotation marks.
**Example:** Using the JPTRANS Function

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

For a b c, the result is АВС.

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

For А В С, the result is а б с.

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

For aаБbCc123, the result is АаBбCбC 123.

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

For !@#$％、．？, the result is ！＠＄％、．？

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

For 「ハ・スホ・ル」, the result is 「ベースボール。」

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

For A a B b C c 1 2 3, the result is AаБbCc123.

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

For !@#$％、．？, the result is !@#$％、．？

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

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

```
JPTRANS('ZNHNSPACE', 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). We have the following EXTRA rule for those special cases.
  
  HNZNSIGN:
  - Double Quote " (U+0022) -> Fullwidth Right Double Quote ” (U+201D)
  - Single Quote ' (U+0027) -> Fullwidth Right Single Quote ’ (U+2019)
  - Comma , (U+002C) -> Fullwidth Ideographic Comma (U+3001)
  - Full Stop . (U+002E) -> Fullwidth Ideographic Full Stop ? (U+3002)
  - Backslash \ (U+005C) -> Fullwidth Backslash \ (U+FF3C)
  - Halfwidth Left Corner Bracket (U+FF62) -> Fullwidth Left Corner Bracket (U+300C)
  - Halfwidth Right Corner Bracket (U+FF63) -> Fullwidth Right Corner 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)
5. Character Functions for DBCS Code Pages

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

- HNZNKANA and ZNHNKANA focus on the conversion of Katakana
  They convert not only letters but also punctuation symbols on the following list:
  - Fullwidth Ideographic Comma (U+3001) <-> Halfwidth Ideographic Comma (U+FF64)
  - Fullwidth Ideographic Full Stop (U+3002) <-> Halfwidth Ideographic Full Stop (U+FF61)
  - Fullwidth Left Corner Bracket (U+300C) <-> Halfwidth Left Corner Braket (U+FF62)
  - Fullwidth Right Corner Bracket (U+300D) <-> Halfwidth Right Corner Bracket (U+FF63)
  - Fullwidth Middle Dot · (U+30FB) <-> Halfwidth Katakana Middle Dot · (U+FF65)
  - Fullwidth Prolonged Sound (U+30FC) <-> Halfwidth Prolonged Sound (U+FF70)

- JPTRANS can be nested for multiple conversions.
  For example, text data may contain fullwidth numbers and fullwidth symbols. In some situations, they should be cleaned up for ASCII numbers and symbols.

For バンゴウ # 1 2 3 , the result is バンゴウ#123

JPTRANS('ZHNHALPHA', 20, JPTRANS('ZHNSIGN', 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.
Maintain-specific Character Functions

Character functions manipulate alphanumeric fields or character strings. The functions in this topic are available only in the Maintain language. There are additional character functions that are available in both the reporting and Maintain languages. For information on these functions, see Character Functions on page 67.

Topics:

- CHAR2INT: Translating a Character Into an Integer Value
- INT2CHAR: Translating an Integer Value Into a Character
- LCWORD and LCWORD2: Converting a Character String to Mixed-Case
- LENGTH: Determining the Length of a Character String
- LJUST: Left-Justifying a Character String (Maintain)
- LOWER: Converting a Character String to Lowercase
- MASK: Extracting or Adding Characters
- MNTGETTOK: Extracting Tokens From a String Function
- NLSCHR: Converting Characters From the Native English Code Page
- OVRLAY: Overlaying a Character String (Maintain)
- POSIT: Finding the Beginning of a Substring (Maintain)
- RJUST: Right-Justifying a Character String (Maintain)
- SELECTS: Decoding a Value From a Stack
- STRAN: Substituting One Substring for Another
- STRCMP: Comparing Character Strings
- STRICMP: Comparing Character Strings and Ignoring Case
- STRNCMP: Comparing Character Substrings
- STRTOKEN: Extracting a Substring Based on Delimiters
- TRIM: Removing Trailing Occurrences (Maintain)
- TRIMLEN: Determining the Length of a String Excluding Trailing Spaces
- UPCASE: Converting Text to Uppercase (Maintain)
CHAR2INT: Translating a Character Into an Integer Value

**How to:**
Translate a Character Into an Integer Value

The CHAR2INT function translates an ASCII or EBCDIC character to the integer value it represents, depending on the operating system.

**Syntax:**

```
CHAR2INT("character")
```

where:

- `character` is the ASCII or EBCDIC character to translate into its integer value.

**Example:**

CHAR2INT translates the character X into its integer equivalent.

```
MAINTAIN
INT/I3=CHAR2INT("X");
type "INT IS <INT";
END
```

On an ASCII platform, the integer value would be 120.
On an EBCDIC platform, the integer value would be 231.

INT2CHAR: Translating an Integer Value Into a Character

**How to:**
Translate an Integer Value Into a Character

The INT2CHAR function translates an integer into the equivalent ASCII or EBCDIC character, depending on the operating system.
**Syntax:** How to Translate an Integer Value Into a Character

```
INT2CHAR(value)
```

where:

- `value`
  - Is the integer to translate into its equivalent ASCII or EBCDIC character.

**Example:** Translating an Integer Value Into a Character

INT2CHAR translates the integer value 93 into its character equivalent.

```
MAINTAIN
CHAR/A1=INT2CHAR(93);
TYPE "CHAR IS <CHAR";
END
```

On an ASCII platform, the result would be a right bracket (]). On an EBCDIC platform, the result would be a right parenthesis.

**LCWORD and LCWORD2: Converting a Character String to Mixed-Case**

**How to:**

Convert a Character String to Mixed-Case

The LCWORD and LCWORD2 functions convert the letters in a character string to mixed-case. These functions convert character strings in the following way:

- **LCWORD.** 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. For example, O’CONNOR is converted to O’Connor and JACK’S to Jack’S.

  If LCWORD encounters a number in the character string, it treats it as an uppercase character and continues to convert the following alphabetic characters to lowercase.

- **LCWORD2.** Converts every alphanumeric character to lowercase except the first letter of each new word. LCWORD2 leaves any character after a single quotation mark as uppercase, except that when there is at least one non-blank character before the quote and just one character followed by either the end of the string or a space immediately after the quote, the next letter is converted to lowercase. For example, 'O’CONNOR' would be changed to 'O’Connor,' and JACK'S would be changed to Jack’s.

To use these functions, you must import the function library MNTUWS. For information on importing this library, see *Accessing and Calling a Function* on page 43.
There is also an LCWORD function available for both the reporting and Maintain languages. For information on this function, see *Character Functions* on page 67.

**Syntax: How to Convert a Character String to Mixed-Case**

```
{LCWORD|LCWORD2}(string)
```

where:

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

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

LCWORD and LCWORD2 convert the string O'CONNOR to mixed-case:

```
MAINTAIN FILE CAR
MODULE IMPORT (MNTUWS)
COMPUTE MYVAL1/A10="O'CONNOR";
   COMPUTE LC1/A10 = LCWORD(MYVAL1);
   COMPUTE LC2/A10 = LCWORD2(MYVAL1);
   TYPE "<<MYVAL1  <<LC1  <<LC2"
END
```

The output is:

```
MYVAL1 LC1 LC2
------- --- ---
O'CONNOR O'Connor O'connor
```

**LENGTH: Determining the Length of a Character String**

**How to:**

Determine the Length of a Character String

The LENGTH function determines the length of a character string, including trailing spaces.
**Syntax:** How to Determine the Length of a Character String

LENGTH(string)

where:

string

Alphanumeric

Is the character string whose length is to be found, or a temporary field that contains the string.

**Example:** Determining the Length of a Character String

LENGTH determines the length of a variable in COUNTRY:

MAINTAIN FILE CAR
MODULE IMPORT (MNTUWS)
NEXT COUNTRY INTO STK1
COMPUTE LEN/I3 = LENGTH(STK1(1).COUNTRY);
TYPE "<STK1(1).COUNTRY HAS A LENGTH OF <<LEN"
END

The result is:

ENGLAND HAS A LENGTH OF 10

**LJUST: Left-Justifying a Character String (Maintain)**

**How to:** Left-Justify a Character String

The LJUST function left-justifies a character string within a field. All leading spaces are removed.

LJUST will not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item.

To use this function, you must import the function library MNTUWS. For information on importing this library see Accessing and Calling a Function on page 43.

There is also an LJUST function available for the reporting language. For information on this function, see Character Functions on page 67.
Syntax: **How to Left-Justify a Character String**

LJUST(*string*)

where:

*string*

Alphanumeric

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

**LOWER: Converting a Character String to Lowercase**

**How to:**

Convert a Character String to Lowercase

The LOWER function converts a character string to lowercase. To use this function, you must import the function library MNTUWS. For more information on importing this library see *Accessing and Calling a Function* on page 43.

Syntax: **How to Convert a Character String to Lowercase**

LOWER(*string*)

where:

*string*

Alphanumeric

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

**MASK: Extracting or Adding Characters**

**How to:**

Extract or Add Characters

The MASK function extracts characters from or adds characters to an alphanumeric string. It can extract a substring from different parts of the parent string, and can insert characters from a parent string into another substring. For example, it can extract the first two characters and the last two characters of a string to form a single substring.
MASK works by comparing the characters in a mask to the characters in a source field. When it encounters a 9 in the mask, MASK copies the corresponding character from the source field to the new field. When it encounters a dollar sign in the mask, MASK ignores the corresponding character in the source field. When it encounters any other character in the mask, MASK copies that character to the corresponding position in the new field.

MASK replaces the masking functionality of the EDIT function that is available in the reporting language.

**Syntax:**

**How to Extract or Add Characters**

MASK(\textit{fieldname}, '\textit{mask}')

where:

\textit{fieldname}

Is the source field.

\textit{mask}

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

**Example:**

**Extracting a Character From a Field**

MASK extracts the first initial from the FIRST_NAME field:

\texttt{MASK(FIRST\_NAME, '9$$$$$$')} \\

The following are sample values for FIRST\_NAME and the values for the result of the MASK function:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>MASK_FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARY</td>
<td>M</td>
</tr>
<tr>
<td>DIANE</td>
<td>D</td>
</tr>
<tr>
<td>JOHN</td>
<td>J</td>
</tr>
<tr>
<td>ROSEMARIE</td>
<td>R</td>
</tr>
<tr>
<td>MARY</td>
<td>M</td>
</tr>
<tr>
<td>BARBARA</td>
<td>B</td>
</tr>
</tbody>
</table>

**Example:**

**Adding Dashes to a Field**

MASK adds dashes to the EMP\_ID field:

\texttt{MASK(EMP\_ID, '999-99-9999')}
The following are sample values for EMP_ID and the values for the result of the MASK function:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>MASK_EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>112847612</td>
<td>112-84-7612</td>
</tr>
<tr>
<td>117593129</td>
<td>117-59-3129</td>
</tr>
<tr>
<td>219984371</td>
<td>219-98-4371</td>
</tr>
<tr>
<td>326179357</td>
<td>326-17-9357</td>
</tr>
<tr>
<td>543729165</td>
<td>543-72-9165</td>
</tr>
<tr>
<td>818692173</td>
<td>818-69-2173</td>
</tr>
</tbody>
</table>

**MNTGETTOK: Extracting Tokens From a String Function**

**How to:**
Extract a Substring (Token)

The Maintain function MNTGETTOK divides a character string into substrings, called tokens. In order to use MNTGETTOK, the data must have a specific character called a delimiter that occurs in the string and separates the string into tokens. MNTGETTOK returns the token specified by the `token_number` argument.

For example, you can use MNTGETTOK to extract individual values from a list separated by semi-colons, by designating the semi-colon as the delimiter.

To use this function, you must import the function library MNTUWS.

**Note:**
- The Maintain function called strtok() returns only the first token from a string.
- MNTGETTOK can work with variable length character strings (format A0).

**Syntax:**

**How to Extract a Substring (Token)**

```plaintext
Module Import(mntuws)
MNTGETTOK(infield,"delim",token_number)
```

where:
- **infield**
  - Alphanumeric
  - Is the field containing the original character string or a character string enclosed in single or double quotation marks.
- **delim**
  - Alphanumeric
Is the delimiter in the parent string enclosed in single or double quotation marks. If you specify more than one character, only the first character is used. The delimiter is not included in the token.

**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.

**Example: Extracting Tokens From a String**

MNTGETTOK extracts tokens from the variable length character string SKILLSTRING and stores the result in the variable length character string TOKENX. The delimiter is a blank space. The token number is based on the value of the counter variable i, which increments with each pass through the Repeat loop:

```plaintext
MAINTAIN
MODULE IMPORT(MNTUWS)
SKILLSTRING/A0="Typing Steno Filing Bkkping"
COMPUTE i/i2 = 1;
TYPE "Job skills required are:"
REPEAT 6
COMPUTE TOKENX/A0=MNTGETTOK(SKILLSTRING, ' ', i);
TYPE "<<TOKENX"
COMPUTE i = i+1;
ENDREPEAT
END

The output is:

Job skills required are:
Typing
Steno
Filing
Bkkping
```
Example: Extracting the Zip Code From an Address

The following procedure against the EMPLOYEE data source retrieves the EMPINFO segment and the first instance of ADDRESS_LN3 for each employee, then extracts the last token (zip code) from ADDRESS_LN3:

```
MAINTAIN FILE EMPLOYEE
MODULE IMPORT(MNTUWS)
REPEAT ALL;
NEXT EMP_ID INTO ESTACK
IF FOCFETCH NE 0 THEN GOTO EXITREPEAT;
NEXT ADDRESS_LN3 INTO ASTACK
TYPE "<<ESTACK.FIRST_NAME "<<ESTACK.LAST_NAME";
TYPE "<<ASTACK.ADDRESS_LN3";
COMPUTE ZIP/A0=MNTGETTOK(ASTACK.ADDRESS_LN3, " ", -1 );
TYPE "ZIP CODE IS: <<ZIP";
TYPE " ";
ENDREPEAT
END
```
The output is:

ALFRED STEVENS
NEW YORK NY 10001
ZIP CODE IS: 10001

MARY SMITH
NEW YORK NY 10001
ZIP CODE IS: 10001

DIANE JONES
NEW YORK NY 10001
ZIP CODE IS: 10001

RICHARD SMITH
NEW YORK NY 10001
ZIP CODE IS: 10001

JOHN BANNING
FREEPORT NY 11520
ZIP CODE IS: 11520

JOAN IRVING
NEW YORK NY 10001
ZIP CODE IS: 10001

ANTHONY ROMANS
NEW YORK NY 10001
ZIP CODE IS: 10001

JOHN MCCOY
NEW YORK NY 10001
ZIP CODE IS: 10001

ROSEMARIE BLACKWOOD
NEW YORK NY 10001
ZIP CODE IS: 10001

ROGER MCKNIGHT
NEW YORK NY 10001
ZIP CODE IS: 10001

MARY GREENSPAN
NEW YORK NY 10001
ZIP CODE IS: 10001

BARBARA CROSS
NEW YORK NY 10001
ZIP CODE IS: 10001
NLSCHR: Converting Characters From the Native English Code Page

**How to:**

Convert Characters From the Native English Code Page

NLSCHR converts a character from the native English code page to the running code page. This is useful when hosting Web applications on an EBCDIC host with non-English code pages.

**Syntax:**

NLSCHR("character")

where:

character

Is the character being converted from the native English code page.

**Example:**

NLSCHR forces the dollar sign to appear whenever the variable ADOLLAR is used, regardless of the code page being run.

```
MAINTAIN
ADOLLAR/A1=NLSCHR("$"halt
.
.
.`
END
```

OVRLAY: Overlaying a Character String (Maintain)

**How to:**

Overlay a Character String

The OVRLAY function overlays a base character string with a substring.

To use this function, you must import the function library MNTUWS. For information on importing this library, see *Accessing and Calling a Function* on page 43.

There is also an OVRLAY function available for the reporting language. For information on this function, see *Character Functions* on page 67.
**Syntax:** How to Overlay a Character String

OVRLAY(string1, string2, position)

where:

string1
- Alphanumeric
- Is the base character string.

string2
- Alphanumeric
- Is the substring that will overlay string1.

position
- Integer
- Is the position in the base string at which the overlay begins.

**Example:** Overlaying a Character String

OVRLAY replaces the letters MCA in the MOVIECODE field with MHD:

```
MAINTAIN FILE movies
Module Import (mntuws);
Case Top
Infer moviecode into MCASTK
Compute MCASTK.NEWCODE/A6;
For all next Moviecode into stk1
Stack copy from stk1 into MCASTK
  where moviecode contains 'MCA';
Compute i/i2=1;
Type "Original Code   New Code"
repeat mcastk.Foccount
  Compute MCASTK(i).Newcode = OVRLAY(MCASTK(i).MOVIECODE, 'MHD', 4);
  Type " <<MCASTK(i).moviecode          <<MCASTK(I).NEWCODE"
  Compute i=i+1;
endrepeat
EndCase
END
```
The following are sample values for MOVIECODE and the values for the result of the OVRLAY function:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>New Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001MCA</td>
<td>001MHD</td>
</tr>
<tr>
<td>081MCA</td>
<td>081MHD</td>
</tr>
<tr>
<td>082MCA</td>
<td>082MHD</td>
</tr>
<tr>
<td>161MCA</td>
<td>161MHD</td>
</tr>
<tr>
<td>196MCA</td>
<td>196MHD</td>
</tr>
<tr>
<td>530MCA</td>
<td>530MHD</td>
</tr>
<tr>
<td>550MCA</td>
<td>550MHD</td>
</tr>
<tr>
<td>883MCA</td>
<td>883MHD</td>
</tr>
</tbody>
</table>

**POSIT: Finding the Beginning of a Substring (Maintain)**

**How to:**

Find the Beginning of a Substring

The POSIT 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.

To use this function, you must import the function library MNTUWS. For information on importing this library see *Accessing and Calling a Function* on page 43.

There is also a POSIT function available for the reporting language. For information on this function, see *POSIT: Finding the Beginning of a Substring* on page 104.

**Syntax:**

How to Find the Beginning of a Substring

The POSIT function is defined as:

\[
\text{POSIT}(parent, \text{substring})
\]

where:

- **parent**
  - Alphanumeric
  - Is the parent string.

- **substring**
  - Alphanumeric
  - Is the substring for which to find the position.
**Example:** Finding the Beginning of a Substring

POSIT displays all movie titles containing the word ROOF and the starting position of the ROOF string:

```plaintext
MAINTAIN FILE movies
Module Import (mntuws);
Case Top
For all next Moviecode into stk1
  Where Title Contains 'ROOF';
Compute i/i2=1;
type "    Title       Start Position of word ROOF"
repeat stk1.Foccount
  Compute STK1(i).POS/I3 = POSIT(STK1(I).TITLE, 'ROOF');
  Type "  <STK1(i).Title  <<STK1(I).pos"
Compute i=i+1;
endrepeat
EndCase
END
```

The following are sample values for MOVIECODE and values for the result of the POSIT function:

<table>
<thead>
<tr>
<th>Title</th>
<th>Start Position of word ROOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIDDLER ON THE ROOF</td>
<td>16</td>
</tr>
<tr>
<td>CAT ON A HOT TIN ROOF</td>
<td>18</td>
</tr>
</tbody>
</table>

**RJUST: Right-Justifying a Character String (Maintain)**

**How to:**

Right-Justify 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.

RJUST does not have any visible effect in a report that uses StyleSheets (SET STYLE=ON) unless you center the item. Also, if you use RJUST on a platform on which StyleSheets are turned on by default, issue SET STYLE=OFF before running the request.

There is also an RJUST function available for the reporting language. For information on this function, see *[RJUST: Right-Justifying a Character String]* on page 108.
**Syntax:**  
How to Right-Justify a Character String

\[ \text{RJUST} (\text{string}, \text{length}, \text{char}) \]

where:

- **string**  
  Is the character string, or a temporary field that contains the string.

- **length**  
  Is the length, in characters, of the result. If this argument is less than the length of \text{string}, RJUST trims \text{string} from right to left. If this argument is zero, RJUST returns a variable length string of length zero.

- **char**  
  Is the character with which to pad the character string and right-justify it. RJUST uses \text{char} only when \text{length} is greater than the length of \text{string}.

**SELECTS: Decoding a Value From a Stack**

**How to:**

Decode a Value From a Stack

The SELECTS function decodes a value from a stack.

**Syntax:**  
How to Decode a Value From a Stack

\[ \text{target} \ \text{SELECTS} \ (\text{code}, \text{result}, \text{code}, \text{result}, \ldots \ [\text{ELSE} \ \text{default}]) \]

where:

- **target**  
  Is a valid expression. It can be either a field name or a variable that resolves to a single stack cell.

- **code**  
  Is the value SELECTS searches for. Once the value is found, the input expression is assigned the corresponding result. The comma between the code and result is optional.

- **result**  
  Is the value assigned when the input expression has the corresponding code.

- **default**  
  Is the value to be assigned if the code is not found among the list of codes. If the default is omitted, a space or zero is assigned to non-matching codes.
**Example:** Decoding Values With SELECTS

The following computes a user-defined field based on the values in a stack:

```
COMPUTE Square = Stk(Cnt).Number SELECTS (1 1, 2 4, 3 9);
```

Because SELECTS is a binary operator, it can also be used in an expression:

```
COMPUTE Square_Plus = Stk(Cnt).Number SELECTS (1 1, 2 4, 3 9) +1;
```

**Example:** Decoding a Value From a Stack

The following example uses MASK to extract the first character of the field CURR_JOBCODE in the EMPLOYEE file. Then SELECTS creates a value for the field JOB_CATEGORY:

```
MAINTAIN FILE Employee
Case Top
  FOR ALL NEXT EMPINFO.EMP_ID INTO EmpStack;
  COMPUTE
    DEPX_CODE/A1 = MASK(EmpStack().CURR_JOBCODE,'9$$');
    JOB_CATEGORY/A15 = DEPX_CODE SELECTS (A 'ADMINISTRATIVE'
      B 'DATA PROCESSING')
  ENDCase
END
```

The following table shows sample values for CURR_JOBCODE and the corresponding values for JOB_CATEGORY:

<table>
<thead>
<tr>
<th>CURR_JOBCODE</th>
<th>JOB_CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>A07</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>A15</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>A17</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>B02</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>B03</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>B04</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>B14</td>
<td>DATA PROCESSING</td>
</tr>
</tbody>
</table>

**STRAN: Substituting One Substring for Another**

**How to:**

Substitute a Substring

The STRAN function substitutes a substring for another substring in a character string. STRAN enables you to edit part of a character string without replacing the field entirely.

To use this function, import the function library MNTUWS. For more information on importing this library see *Calling a Function* on page 44.
**Syntax:** How to Substitute a Substring

STRAN(string, substr1, substr2)

where:

*string*  
Alphanumeric  
Is the character string into which you want to substitute one substring for another, or a temporary field that contains the string.

*substr1*  
Alphanumeric  
Is the substring to replace.

*substr2*  
Alphanumeric  
Is the substring to insert in place of *substr1*.

**Example:** Substituting One String for Another

STRAN replaces the word DOOR with the word Seater in the MODEL field:

```
MAINTAIN FILE CAR
MODULE IMPORT (MNTUWS);
FOR ALL NEXT COUNTRY CAR MODEL INTO XSTK
   WHERE MODEL CONTAINS 'DOOR'
   COMPUTE XSTK.NEWMOD/A24;
   COMPUTE I/I2=1;
REPEAT XSTK.FOCOUNT
   COMPUTE XSTK(I).NEWMOD=STRAN(XSTK(I).MODEL, 'DOOR', 'SEATER');
   TYPE "<<XSTK(I).CAR  <<XSTK(I).MODEL  <<XSTK(I).NEWMOD"
   COMPUTE I=I+1;
ENDREPEAT
END
```
The following are sample values for MODEL and values for the result of the STRAN function:

<table>
<thead>
<tr>
<th>CAR</th>
<th>MODEL</th>
<th>STRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEUGEOT</td>
<td>504 4 DOOR</td>
<td>504 4 SEATER</td>
</tr>
<tr>
<td>ALFA ROMEO</td>
<td>2000 4 DOOR BERLINA</td>
<td>2000 4 SEATER BERLINA</td>
</tr>
<tr>
<td>MASERATI</td>
<td>DORA 2 DOOR</td>
<td>DORA 2 SEATER</td>
</tr>
<tr>
<td>DATSUN</td>
<td>B210 2 DOOR AUTO</td>
<td>B210 2 SEATER AUTO</td>
</tr>
<tr>
<td>TOYOTA</td>
<td>COROLLA 4 DOOR DIX AUTO</td>
<td>COROLLA 4 SEATER DIX AUT</td>
</tr>
<tr>
<td>AUDI</td>
<td>100 LS 2 DOOR AUTO</td>
<td>100 LS 2 SEATER AUTO</td>
</tr>
<tr>
<td>BMW</td>
<td>2002 2 DOOR AUTO</td>
<td>2002 2 SEATER AUTO</td>
</tr>
<tr>
<td>BMW</td>
<td>3.0 SI 4 DOOR AUTO</td>
<td>3.0 SI 4 SEATER AUTO</td>
</tr>
<tr>
<td>BMW</td>
<td>530I 4 DOOR AUTO</td>
<td>530I 4 SEATER</td>
</tr>
</tbody>
</table>

**STRCMP: Comparing Character Strings**

**How to:**

Compare Character Strings

The STRCMP function compares two character strings using the EBCDIC or ASCII collating sequence.

- If the first string is less than the second string, STRCMP returns a negative value.
- If the first string is greater than the second string, STRCMP returns a positive value.
- If the first string is equal to the second string, STRCMP returns zero.

**Syntax:**

**How to Compare Character Strings**

```plaintext
STRCMP(string1, string2)
```

where:

- `string1, string2`
  - Alphanumeric
  - Are the strings to compare, or temporary fields that contain the strings.
Example: Comparing Character Strings

STRCMP compares the length of two fields:

MAINTAIN
COMPUTE STR1/A20 = 'STRING IS LONG';
    STR2/A20 = 'STRING IS LONGER';
COMPUTE DIF/I3= STRCMP (STR1, STR2);
TYPE "STR1 = <<STR1"
TYPE "STR2 = <<STR2"
IF DIF LT 0 THEN TYPE "STR2 IS GREATER THAN STR1"
ELSE IF DIF GT 0 THEN TYPE "STR2 IS LESS THAN STR1"
ELSE IF DIF EQ 0 THEN TYPE "STR2 EQUALS STR1"
TYPE " "
COMPUTE STR3/A20 = 'STRING IS LONGEST';
    STR4/A20 = 'STRING IS LONG';
TYPE "STR3 = <<STR3"
TYPE "STR4 = <<STR4"
COMPUTE DIF= STRCMP (STR3, STR4);
IF DIF LT 0 THEN TYPE "STR4 IS GREATER THAN STR3"
ELSE IF DIF GT 0 THEN TYPE "STR4 IS LESS THAN STR3"
ELSE IF DIF EQ 0 THEN TYPE "STR4 EQUALS STR3"
TYPE " "
COMPUTE DIF= STRCMP (STR1, STR4);
IF DIF LT 0 THEN TYPE "STR1 IS GREATER THAN STR4"
ELSE IF DIF GT 0 THEN TYPE "STR1 IS LESS THAN STR4"
ELSE IF DIF EQ 0 THEN TYPE "STR1 EQUALS STR4"
END

The result is:

STR1 = STRING IS LONG
STR2 = STRING IS LONGER
STR2 IS GREATER THAN STR1
STR3 = STRING IS LONGEST
STR4 = STRING IS LONG
STR4 IS LESS THAN STR3
STR1 EQUALS STR4
STRICMP: Comparing Character Strings and Ignoring Case

**How to:**
Compare Character Strings and Ignore Case

The STRICMP function compares two character strings using the EBCDIC or ASCII collating sequence, but ignores case differences.

- If the first string is less than the second string, STRICMP returns a negative value.
- If the first string is greater than the second string, STRICMP returns a positive value.
- If the first string is equal to the second string, STRICMP returns zero.

**Syntax:**

```
STRICMP(string1, string2)
```

where:

- `string1, string2`
  Alphanumeric

  Are the strings to compare, or temporary fields that contain the strings.

STRNCMP: Comparing Character Substrings

**How to:**
Compare Character Substrings

The STRNCMP function compares a specified number of characters in two character strings starting at the beginning of the strings using the EBCDIC or ASCII collating sequence.

- If the first string is less than the second string, STRNCMP returns a negative value.
- If the first string is greater than the second string, STRNCMP returns a positive value.
- If the first string is equal to the second string, STRNCMP returns zero.
**Syntax: How to Compare Character Substrings**

```
STRNCMP(string1, string2, number)
```

where:

- **string1, string2**
  - Alphanumeric
  - Are the strings that contain the substrings to compare.

- **number**
  - Integer
  - Is the number of characters to compare in string1 and string2.

**STRTOKEN: Extracting a Substring Based on Delimiters**

**How to:**

**Extract a Substring**

The STROKE function returns a substring, that consists of a string's characters from the beginning of a string to a specified character, called a delimiter.

To use this function, you must import the function library MNTUWS. For more information on importing this library see *Calling a Function* on page 44.

**Syntax: How to Extract a Substring**

```
STRTOKEN(string, delimiters)
```

where:

- **string**
  - Alphanumeric
  - Is the character string, or a variable that contains the string enclosed in double quotation marks.

- **delimiters**
  - Alphanumeric
  - Is a character string, or variable enclosed in double quotation marks that contains a list of delimiters. Separate the delimiters with semicolons.
**Example:** Extracting a Substring

`STRTOKEN` returns a substring of the first five `STREET` values in the VIDEOTRK data source based on the delimiters period, space, or asterisk.

```plaintext
MAINTAIN FILE VIDEOTRK
MODULE IMPORT (MNTUWS);
FOR ALL NEXT CUSTID INTO CSTACK;
COMPUTE CNT/I5 = 1;
TYPE " ";
REPEAT WHILE CNT LE 5;
COMPUTE SUBSTREET/A20 = STRTOKEN(CSTACK(CNT).STREET,".; *,*")));
TYPE " STREET = <CSTACK(CNT).STREET"
TYPE " SUBSTREET = <SUBSTREET "
COMPUTE CNT = CNT +1;
ENDREPEAT
END
```

The output is:

- STREET = 86 ELLIOTT AVE.
- SUBSTREET = 86
- STREET = 7 DAVENPORT LA.
- SUBSTREET = 7
- STREET = 8 MAGNOLIA LA.
- SUBSTREET = 8
- STREET = 35 POWELL ST.
- SUBSTREET = 35
- STREET = 10 COW LA.
- SUBSTREET = 10

**SUBSTR: Extracting a Substring (Maintain)**

**How to:**

Extract a Substring

The `SUBSTR` function extracts a substring based on where it begins and its length in the parent string. `SUBSTR` can vary the position of the substring depending on the values of other fields.

There is also a `SUBSTR` function available for the reporting language. For information on this function, see *SUBSTR: Extracting a Substring* on page 118.
Syntax:  How to Extract a Substring

\[
\text{SUBSTR}(\text{string, start, length})
\]

where:

- **string**
  - Alphanumeric
  - Is the parent string enclosed in single quotation marks, or a field or variable containing the character string.

- **start**
  - Integer
  - Is the starting position of the substring in the parent string.

- **length**
  - Integer
  - Is the length, in characters, of the substring.

Example: Extracting the First Character of a String in Maintain

SUBSTR extracts the first letter of FIRST_NAME, combines it with LAST_NAME, and stores the result in UID:

```maintain
MAINTAIN FILE EMPLOYEE
CASE TOP
INFER EMP_ID FIRST_NAME LAST_NAME INTO ADDSTACK
COMPUTE UID/A9 = SUBSTR(ADDSTACK().FIRST_NAME,1,1) || ADDSTACK().LAST_NAME;
ENDCASE
END
```

The following table shows sample values for FIRST_NAME and LAST_NAME, and the corresponding values for UID:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOE</td>
<td>SMITH</td>
<td>JSMITH</td>
</tr>
<tr>
<td>SAM</td>
<td>JONES</td>
<td>SJONES</td>
</tr>
<tr>
<td>TERRI</td>
<td>WHITE</td>
<td>TWHITE</td>
</tr>
</tbody>
</table>
**TRIM: Removing Trailing Occurrences (Maintain)**

### How to:
Remove Trailing Occurrences

The TRIM function removes trailing occurrences of a pattern within a character string.

There is also a TRIM function available for the reporting language. For information on this function, see *TRIM: Removing Leading and Trailing Occurrences* on page 121.

**Syntax:**

**How to Remove Trailing Occurrences**

```
TRIM(string)
```

where:

- `string`
  - Alphanumeric
    - Is the character string enclosed in single quotation marks, or the field containing the string.

**TRIMLEN: Determining the Length of a String Excluding Trailing Spaces**

### How to:
Determine the Length of a String Excluding Trailing Spaces

The TRIMLEN function determines the length of a character string excluding trailing spaces.

**Syntax:**

**How to Determine the Length of a String Excluding Trailing Spaces**

```
TRIMLEN (string)
```

where:

- `string`
  - Alphanumeric
    - Is the string to be measured.
**Example:** Determining the Length of a String Excluding Trailing Spaces

TRIMLEN determines the length of a field in COUNTRY excluding trailing blanks:

```
MAINTAIN FILE CAR
MODULE IMPORT (MNTUWS)
NEXT COUNTRY INTO STK1
COMPUTE LEN/I3 = LENGTH(STK1(1).COUNTRY);
COMPUTE LEN2/I3 = TRIMLEN(STK1(1).COUNTRY);
TYPE "<STK1(1).COUNTRY HAS A LENGTH OF <LEN2 WITHOUT TRAILING BLANKS"
END
```

The result is:

```
ENGLAND HAS A LENGTH OF 7 WITHOUT TRAILING BLANKS
```

**UPCASE: Converting Text to Uppercase (Maintain)**

**How to:** Convert Text to Uppercase

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

To use this function, you must import the function library MNTUWS. For information on importing this library, see *Calling a Function* on page 44.

There is also an UPCASE function available for the reporting language. For information on this function, see *UPCASE: Converting Text to Uppercase* on page 123.

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

```
UPCASE(string)
```

where:

```
string
```

Alphanumeric

- Is the character string to be converted to uppercase.
Data source and decoding functions search for data source records, retrieve data source records or values, and assign values based on the value of an input field.

The result of a data source function must be stored in a field. The result cannot be stored in a Dialogue Manager variable.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function on page 43.

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

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

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

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

**Syntax:**

**How to Retrieve a Value From a Lookup Data Source**

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

where:

- `look_mf`
  - 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 WebFOCUS or FOCUS. Other types of lookup files can be reused without clearing the DEFINE. They will be cleared automatically when all DEFINE fields are cleared.

- If the lookup field has the MISSING=ON attribute in its Master File and the DEFINE or COMPUTE command specifies MISSING ON, the missing value is returned when the lookup field is missing. Without MISSING ON in both places, the missing value is converted to a default value (blank for an alphanumeric field, zero for a numeric field).

- Source records display on the report output even if they lack a matching record in the lookup file.

- Only real fields in the lookup Master File are valid as lookup and return fields.

- If there are multiple rows in the lookup table where the source field is equal to the lookup field, the first value of the return field is returned.
Example: Retrieving a Value From a Fixed Format Sequential File in a TABLE Request

The following procedure creates a fixed format sequential file named GSALE from the GGSALES data source. The fields in this file are PRODUCT (product description), CATEGORY (product category), and PCD (product code). The file is sorted on the PCD field:

```plaintext
SET ASNAMES = ON
TABLE FILE GGSALES
SUM PRODUCT CATEGORY
BY PCD
ON TABLE HOLD AS GSALE FORMAT ALPHA
END
```

The following Master File is generated as a result of the HOLD command:

```plaintext
FILENAME=GSALE, SUFFIX=FIX , $
    SEGMENT=GSALE, SEGTYPE=S1, $
        FIELDNAME=PCD, ALIAS=E01, USAGE=A04, ACTUAL=A04, $
        FIELDNAME=PRODUCT, ALIAS=E02, USAGE=A16, ACTUAL=A16, $
        FIELDNAME=CATEGORY, ALIAS=E03, USAGE=A11, ACTUAL=A11, $
```

The following TABLE request against the GGPRODS data source, sorts the report on the field that matches the key field in the lookup file. It retrieves the value of the CATEGORY field from the GSALE lookup file by matching on the product code and product description fields. Note that the DEFINE FILE command is cleared at the end of the request:

```plaintext
DEFINE FILE GGPRODS
PCAT/A11 MISSING ON = DB_LOOKUP(GSALE, PRODUCT_ID, PCD, PRODUCT_DESCRIPTION, PRODUCT, CATEGORY);
END
TABLE FILE GGPRODS
PRINT PRODUCT_DESCRIPTION PCAT
BY PRODUCT_ID
END
DEFINE FILE GGPRODS CLEAR
END
```
Because the GSALE Master File does not define the CATEGORY field with the MISSING=ON attribute, the PCAT column displays a blank in those rows that have no matching record in the lookup file:

<table>
<thead>
<tr>
<th>Product Code</th>
<th>Product Code</th>
<th>PCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B141</td>
<td>Hazelnut</td>
<td></td>
</tr>
<tr>
<td>B142</td>
<td>French Roast</td>
<td></td>
</tr>
<tr>
<td>B144</td>
<td>Kona</td>
<td></td>
</tr>
<tr>
<td>F101</td>
<td>Scone</td>
<td>Food</td>
</tr>
<tr>
<td>F102</td>
<td>Biscotti</td>
<td>Food</td>
</tr>
<tr>
<td>F103</td>
<td>Croissant</td>
<td>Food</td>
</tr>
<tr>
<td>G100</td>
<td>Mug</td>
<td>Gifts</td>
</tr>
<tr>
<td>G104</td>
<td>Thermos</td>
<td>Gifts</td>
</tr>
<tr>
<td>G110</td>
<td>Coffee Grinder</td>
<td>Gifts</td>
</tr>
<tr>
<td>G121</td>
<td>Coffee Pot</td>
<td>Gifts</td>
</tr>
</tbody>
</table>

If you add the MISSING=ON attribute to the CATEGORY field in the GSALE Master File, the PCAT column displays a missing data symbol in rows that do not have a matching record in the lookup file:

<table>
<thead>
<tr>
<th>Product Code</th>
<th>Product Code</th>
<th>PCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B141</td>
<td>Hazelnut</td>
<td>.</td>
</tr>
<tr>
<td>B142</td>
<td>French Roast</td>
<td>.</td>
</tr>
<tr>
<td>B144</td>
<td>Kona</td>
<td>.</td>
</tr>
<tr>
<td>F101</td>
<td>Scone</td>
<td>Food</td>
</tr>
<tr>
<td>F102</td>
<td>Biscotti</td>
<td>Food</td>
</tr>
<tr>
<td>F103</td>
<td>Croissant</td>
<td>Food</td>
</tr>
<tr>
<td>G100</td>
<td>Mug</td>
<td>Gifts</td>
</tr>
<tr>
<td>G104</td>
<td>Thermos</td>
<td>Gifts</td>
</tr>
<tr>
<td>G110</td>
<td>Coffee Grinder</td>
<td>Gifts</td>
</tr>
<tr>
<td>G121</td>
<td>Coffee Pot</td>
<td>Gifts</td>
</tr>
</tbody>
</table>

**DECODE: Decoding Values**

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

**Reference:**
Guidelines for Reading Values From a File

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.

The use of DECODE with Maintain is limited. For information on decoding values with subscripted stack values, see SELECTS: Decoding a Value From a Stack on page 166.

**Syntax:**

**How to Supply Values in the Function**

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

where:

*fieldname*

  Alphanumeric or Numeric

  Is the name of the input field.

*code*

  Alphanumeric or Numeric

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

*result*

  Alphanumeric or Numeric

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

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

*default*

  Alphanumeric or Numeric

  Is the value returned as a result for non-matching codes. The format must be the same as the format of result. If you omit a default value, DECODE assigns a blank or zero to non-matching codes.

*filename*

  Alphanumeric

  Is the ddname that points to the file in which code/result pairs are stored. Every record in the file must contain a pair.
You can use up to 40 lines to define the code and result pairs for any given DECODE function, or 39 lines if you also use an ELSE phrase. Use either a comma or blank to separate the code from the result, or one pair from another.

**Note:** DECODE has no output argument.

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

EDIT extracts the first character of the CURR_JOBCODE field, then DECODE returns either ADMINISTRATIVE or DATA PROCESSING depending on the value extracted.

```
TABLE FILE EMPLOYEE
PRINT CURR_JOBCODE AND COMPUTE
DEPX_CODE/A1 = EDIT(CURR_JOBCODE, '9$$'); NOPRINT AND COMPUTE
JOB_CATEGORY/A15 = DECODE DEPX_CODE(A 'ADMINISTRATIVE'
B 'DATA PROCESSING');
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>CURR_JOBCODE</th>
<th>JOB_CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>B04</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>CROSS</td>
<td>A17</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>A07</td>
<td>ADMINISTRATIVE</td>
</tr>
<tr>
<td>JONES</td>
<td>B03</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>MCCOY</td>
<td>B02</td>
<td>DATA PROCESSING</td>
</tr>
<tr>
<td>SMITH</td>
<td>B14</td>
<td>DATA PROCESSING</td>
</tr>
</tbody>
</table>

**Reference:**  Guidelines for Reading Values From a File

- Each record in the file is expected to contain pairs of elements separated by a comma or blank.
- If each record in the file consists of only one element, this element is interpreted as the code, and the result becomes either a blank or zero, as needed.

This makes it possible to use the file to hold screening literals referenced in the screening condition:

```
IF field IS (filename)
```

and as a file of literals for an IF criteria specified in a computational expression. For example:

```
TAKE = DECODE SELECT (filename ELSE 1);
VALUE = IF TAKE IS 0 THEN... ELSE...;
```
TAKE is 0 for SELECT values found in the literal file and 1 in all other cases. The VALUE computation is carried out as if the expression had been:

\[
\text{IF SELECT (filename) THEN... ELSE...;}
\]

- The file can contain up to 32,767 characters in the file.
- All data is interpreted in ASCII format on UNIX, or in EBCDIC format on z/OS, and converted to the USAGE format of the DECODE pairs.
- Leading and trailing blanks are ignored.
- The remainder of each record is ignored and can be used for comments or other data. This convention applies in all cases, except when the file name is HOLD. In that case, the file is presumed to have been created by the HOLD command, which writes fields in the internal format, and the DECODE pairs are interpreted accordingly. In this case, extraneous data in the record is ignored.

**Example:** **Reading DECODE Values From a File**

The following example has two parts. The first part creates a file with a list of IDs and reads the EDUCFILE data source. The second part reads the EMPLOYEE data source and assigns 0 to those employees who have taken classes and 1 to those employees who have not. The HOLD file contains only one column of values; therefore, DECODE assigns the value 0 to an employee whose EMP_ID appears in the file and 1 when EMP_ID does not appear in the file.

```
TABLE FILE EDUCFILE
PRINT EMP_ID
ON TABLE HOLD
END

TABLE FILE EMPLOYEE
PRINT EMP_ID AND LAST_NAME AND FIRST_NAME AND COMPUTE
NOT_IN_LIST/I1 = DECODE EMP_ID(HOLD ELSE 1);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NOT_IN_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>112847612</td>
<td>SMITH</td>
<td>MARY</td>
<td>0</td>
</tr>
<tr>
<td>117593129</td>
<td>JONES</td>
<td>DIANE</td>
<td>0</td>
</tr>
<tr>
<td>219984371</td>
<td>MCCOY</td>
<td>JOHN</td>
<td>1</td>
</tr>
<tr>
<td>326179357</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>0</td>
</tr>
<tr>
<td>543729165</td>
<td>GREENSPAN</td>
<td>MARY</td>
<td>1</td>
</tr>
<tr>
<td>818692173</td>
<td>CROSS</td>
<td>BARBARA</td>
<td>0</td>
</tr>
</tbody>
</table>
FIND: Verifying the Existence of a Value in a Data Source

How to:
Verify the Existence of a Value in a Data Source

Available Languages: MODIFY, Maintain

The FIND function determines if a data value is in a data source field being searched. The function sets a temporary field to 1 (a non-zero value for MODIFY) if the data value is found in the data source field, and to 0 if it is not. FIND does not change the searched file's current database position. A value greater than zero confirms the presence of the data value, not the number of instances in the data source field.

Note: For MODIFY only, the FIND function verifies the existence of an incoming data value in an indexed FOCUS data source field.

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

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

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

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

Syntax: How to Verify the Existence of a Value in a Data Source

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.

For Maintain - the AS field is required and the name must be qualified.

For MODIFY - the AS field must be indexed. If the incoming field and the data source field have the same name, omit this phrase.
file

Is the name of the FOCUS data source.

For Maintain - the IN file is unnecessary since the AS field name is required and must be qualified.

For MODIFY - the IN field must be indexed.

Note:

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

Example: Verifying the Existence of a Value in Another Data Source (Maintain)

In the following example, FIND determines if a data value is found in another data source.

```
MAINTAIN FILE MOVIES AND VIDEOTRK
FOR ALL NEXT MOVIES.MOVIECODE INTO FILMSTK
TYPE "RC SHOULD BE 1 WHERE MOVIECODE EXISTS IN BOTH FILES";
TYPE ""
COMPUTE RC/I1;
COMPUTE I/I1=1;
REPEAT FILMSTK.FOCCOUNT
  COMPUTE RC= FIND(FILMSTK(I).MOVIECODE AS VIDEOTRK.MOVIECODE)
  TYPE "FOR MOVIECODE = <<FILMSTK(I).MOVIECODE , RC = <<RC"
  COMPUTE I=I+1;
ENDREPEAT
END
```

The output is:

RC SHOULD BE 1 WHERE MOVIECODE EXISTS IN BOTH FILES
FOR MOVIECODE = 001MCA, RC = 1
  .
  .
FOR MOVIECODE = 387PLA, RC = 0
  .
  .
FOR MOVIECODE = 963CBS, RC = 1
TRANSACTIONS: COMMITS = 1 ROLLBACKS = 0
SEGMENTS : INCLUDED = 0 UPDATED = 0 DELETED = 0
**Example:** **Verifying the Existence of a Value in the Same Data Source (Maintain)**

In the following example, FIND determines if a data value is found in the same data source.

```
MAINTAIN FILE CAR
COMPUTE RETAIL_COST=31500;
COMPUTE CHECK/I1;
COMPUTE CHECK= FIND (RETAIL_COST);
    IF CHECK = 1 THEN GOTO FOUND1
    ELSE GOTO NOT1;
CASE FOUND1
    TYPE "THERE IS A CAR WITH A RETAIL_COST OF <<RETAIL_COST"
    -* ....
ENDCASE
CASE NOT1
    TYPE "THERE IS NO CAR WITH A RETAIL_COST OF <<RETAIL_COST"
    -* ....
ENDCASE
-* ....
END
```

The output is:

```
THERE IS A CAR WITH A RETAIL_COST OF 31,500
TRANSACTIONS: COMMITS   =    1 ROLLBACKS =    0
SEGMENTS    : INCLUDED  =    0 UPDATED   =    0 DELETED   =    0
```

**Example:** **Verifying the Existence of a Value in an Indexed Field (MODIFY)**

FIND determines if a supplied value in the EMP_ID field is in the EDUCFILE data source. The procedure then displays a message indicating the result of the search.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
COMPUTE
    EDTEST = FIND(EMP_ID IN EDUCFILE);
    MSG/A40 = IF EDTEST NE 0 THEN
        'STUDENT LISTED IN EDUCATION FILE' ELSE
        'STUDENT NOT LISTED IN EDUCATION FILE';
MATCH EMP_ID
    ON NOMATCH TYPE "<MSG"
    ON MATCH TYPE "<MSG"
DATA
```
A sample execution is:

```
> 
EMPLOYEE ON 12/04/2001 AT 12.09.03
DATA FOR TRANSACTION 1

EMP_ID               =
112847612
STUDENT LISTED IN EDUCATION FILE
DATA FOR TRANSACTION 2

EMP_ID               =
219984371
STUDENT NOT LISTED IN EDUCATION FILE
DATA FOR TRANSACTION 3
```

The procedure processes as follows:

1. The procedure prompts you for an employee ID. You enter 112847612.
2. The procedure searches the EDUCFILE data source for the employee ID 112847612. It finds the ID so it prints STUDENT LISTED IN EDUCATION FILE.
3. The procedure prompts you for an employee ID. You enter 219984371.
4. The procedure searches the EDUCFILE data source for the employee ID 219984371. It does not find the ID so it prints STUDENT NOT LISTED IN EDUCATION FILE.

**Example:** **Rejecting a Transaction When a Value Is Not Found (MODIFY)**

The following updates the number of hours an employee spent in class. The VALIDATE command rejects a transaction for an employee whose ID is not found in the EDUCFILE data source, which records class attendance.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
   EDTEST = FIND(EMP_ID IN EDUCFILE);
MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH UPDATE ED_HRS
DATA
```
A sample execution is:

```
> EMPLOYEE ON 12/04/2001 AT 12/26/08
DATA FOR TRANSACTION 1

EMP_ID       = 112847612
ED_HRS     = 7
DATA FOR TRANSACTION 2

EMP_ID       = 219984371
ED_HRS     = 0
(FOC421) TRANS 2 REJECTED INVALID EDTEST
219984371, 0, $
DATA FOR TRANSACTION 3
```

The procedure processes as follows:

1. The procedure prompts you for an employee ID and the number of hours the employee spent in class. You enter the following data:

   ```
   EMP_ID: 112847612
   ED_HRS: 7
   ```

2. The procedure updates the number of hours for the ID 112847612.

3. The procedure prompts you for an employee ID and the number of hours the employee spent in class. You enter the following data:

   ```
   EMP_ID: 219984371
   ED_HRS: 0
   ```

4. The procedure rejects the record for the ID 219984371 because it does not exist in the EDUCFILE data source, and an error message is returned.

**LAST: Retrieving the Preceding Value**

**How to:**

Retrieve the Preceding Value

The LAST function retrieves the preceding value for a field.
The effect of LAST depends on whether it appears in a DEFINE or COMPUTE command:

- In a DEFINE command, the LAST value applies to the previous record retrieved from the data source before sorting takes place.
- In a COMPUTE command, the LAST value applies to the record in the previous line of the internal matrix.

Do not use LAST with the -SET command in Dialogue Manager.

**Syntax:**

**How to Retrieve the Preceding Value**

LAST *fieldname*

where:

*fieldname*

  Alphanumeric or Numeric

  Is the field name.

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

**Example:**

**Retrieving the Preceding Value**

LAST retrieves the previous value of the DEPARTMENT field to determine whether to restart the running total of salaries by department. If the previous value equals the current value, CURR_SAL is added to RUN_TOT to generate a running total of salaries within each department.

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME CURR_SAL AND COMPUTE
RUN_TOT/D12.2M = IF DEPARTMENT EQ LAST DEPARTMENT THEN
  (RUN_TOT + CURR_SAL) ELSE CURR_SAL;
AS 'RUNNING,TOTAL,SALARY'
BY DEPARTMENT SKIP-LINE
END
```
<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>LAST_NAME</th>
<th>CURR_SAL</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>SMITH</td>
<td>$13,200.00</td>
<td>$13,200.00</td>
</tr>
<tr>
<td></td>
<td>JONES</td>
<td>$18,480.00</td>
<td>$31,680.00</td>
</tr>
<tr>
<td></td>
<td>MCCOY</td>
<td>$18,480.00</td>
<td>$50,160.00</td>
</tr>
<tr>
<td></td>
<td>BLACKWOOD</td>
<td>$21,780.00</td>
<td>$71,940.00</td>
</tr>
<tr>
<td></td>
<td>GREENSPAN</td>
<td>$9,000.00</td>
<td>$80,940.00</td>
</tr>
<tr>
<td></td>
<td>CROSS</td>
<td>$27,062.00</td>
<td>$108,002.00</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>STEVENS</td>
<td>$11,000.00</td>
<td>$11,000.00</td>
</tr>
<tr>
<td></td>
<td>SMITH</td>
<td>$9,500.00</td>
<td>$20,500.00</td>
</tr>
<tr>
<td></td>
<td>BANNING</td>
<td>$29,700.00</td>
<td>$50,200.00</td>
</tr>
<tr>
<td></td>
<td>IRVING</td>
<td>$26,862.00</td>
<td>$77,062.00</td>
</tr>
<tr>
<td></td>
<td>ROMANS</td>
<td>$21,120.00</td>
<td>$98,182.00</td>
</tr>
<tr>
<td></td>
<td>MCKNIGHT</td>
<td>$16,100.00</td>
<td>$114,282.00</td>
</tr>
</tbody>
</table>

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

In this section:

Using the Extended LOOKUP Function

**How to:**

Retrieve a Value From a Cross-referenced Data Source

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

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

To modify more than one data source in one request, use the COMBINE command or the Maintain facility.

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

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
  \[ \text{RTN} = \text{LOOKUP(SEG.DATE_ATTEND);} \]

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

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

- If there are no instances of the value, the function sets a return variable to 0. If you use the field specified by LOOKUP in the request, the field assumes a value of blank if alphanumeric and 0 if numeric.

- If there are instances of the value, the function sets the return variable to 1 and retrieves the value of the specified field from the first instance it finds. There can be more than one if the cross-referenced segment type is KM or DKM, or if you specified the ALL keyword in the JOIN command.

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

\[ \text{LOOKUP(field);} \]

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: Reading a Value From a Cross-referenced Data Source

You may need to determine if employees were hired before or after a specific date, for example, January 1, 1982. The employee IDs (EMP_ID) and hire date (HIRE_DATE) are located in the host segment. The following diagram shows the file structure:

![File Structure Diagram]

The request is:

MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
COMPUTE EDTEST = LOOKUP(HIRE_DATE);
    COMPUTE ED_HRS = IF DATE_ENROLL GE 820101 THEN ED_HRS * 1.1
                   ELSE ED_HRS;
MATCH EMP_ID
    ON MATCH UPDATE ED_HRS
    ON NOMATCH REJECT
DATA

A sample execution is:

1. The request prompts you for the employee ID and number of class hours. Enter the ID 117593129 and 10 class hours.
2. LOOKUP locates the first instance in the cross-referenced segment containing the employee ID 117593129. Since the instance exists, the function returns a 1 to the EDTEST variable. This instance lists the enroll date as 821028 (October 28, 1982).

3. LOOKUP retrieves the value 821028 for the DATE_ENROLL field.

4. The COMPUTE command tests the value of DATE_ENROLL. Since October 28, 1982 is after January 1, 1982, the ED_HRS are increased from 10 to 11.

5. The request updates the classroom hours for employee 117593129 with the new value.

**Example:**  Using a Value in a Host Segment to Search a Data Source

You can use a field value in a host segment instance to search a cross-referenced segment. Do the following:

- In the MATCH command that selects the host segment instance, activate the host field with the ACTIVATE command.
- In the same MATCH command, code LOOKUP after the ACTIVATE command.

This request displays the employee ID, date of salary increase, employee name, and the employee position after the raise was granted:

- The employee ID and name (EMP_ID) are in the root segment.
- The date of increase (DAT_INC) is in the descendant host segment.
- The job position is in the cross-referenced segment.
- The shared field is JOBCODE. You never enter a job code; the values are stored in the data source.

The request is:

```sql
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DAT_INC
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DAT_INC
  ON NOMATCH REJECT
  ON MATCH ACTIVATE JOBCODE
  ON MATCH COMPUTE
    RTN = LOOKUP (JOB_DESC);
  ON MATCH TYPE
    "EMPLOYEE ID: <EMP_ID"
    "DATE INCREASE: <DAT_INC"
    "NAME: <D.FIRST_NAME> <D.LAST_NAME"
    "POSITION: <JOB_DESC"
DATA
```
A sample execution is:

1. The request prompts you for the employee ID and date of pay increase. Enter the employee ID 071382660 and the date 820101 (January 1, 1982).

2. The request locates the instance containing the ID 071382660, then locates the child instance containing the date of increase 820101.

3. This child instance contains the job code A07. The ACTIVATE command makes this value available to LOOKUP.

4. LOOKUP locates the job code A07 in the cross-referenced segment. It returns a 1 the RTN variable and retrieves the corresponding job description SECRETARY.

5. The TYPE command displays the values:

```
EMPLOYEE ID:          071382660
DATE INCREASE:        82/01/01
NAME:                 ALFRED STEVENS
POSITION:             SECRETARY
```

Fields retrieved by LOOKUP do not require the D. prefix. FOCUS treats the field values as transaction values.

You may also need to activate the host field if you are using LOOKUP within a NEXT command. This request displays the latest position held by an employee:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH CONTINUE
NEXT DAT_INC
    ON NONEXT REJECT
    ON NEXT ACTIVATE JOB_DESC
    ON NEXT COMPUTE
        RTN = LOOKUP(JOB_DESC);
    ON MATCH TYPE
        "EMPLOYEE ID:        <EMP_ID"
        "DATE OF POSITION:  <DAT_INC"
        "NAME:              <D.FIRST_NAME <D.LAST_NAME"
        "POSITION:          <JOB_DESC"
```

DATA
Example: Using the LOOKUP Function With a VALIDATE Command

When you use LOOKUP, reject transactions containing values for which there is no corresponding instance in the cross-reference segment. To do this, place the function in a VALIDATE command. If the function cannot locate the instance in the cross-referenced segment, it sets the value of the return variable to 0, causing the request to reject the transaction.

The following request updates an employee's classroom hours (ED_HRS). If the employee enrolled in classes on or after January 1, 1982, the request increases the number of classroom hours by 10%. The enrollment dates are stored in a cross-referenced segment (field DATE_ATTEND). The shared field is the employee ID.

The request is as follows:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
  TEST_DATE = LOOKUP(DATE_ENROLL);
COMPUTE
  ED_HRS = IF DATE_ENROLL GE 820101 THEN ED_HRS * 1.1
           ELSE ED_HRS;
MATCH EMP_ID
  ON MATCH UPDATE ED_HRS
  ON NOMATCH REJECT

DATA
```

If an employee record is not found in the cross-referenced segment, that employee never enrolled in a class. The transaction is rejected as an error.

Using the Extended LOOKUP Function

How to:

Use the Extended LOOKUP Function

If the LOOKUP function cannot locate a value of the host field in the cross-referenced segment, use extended syntax to locate the next highest or lowest cross-referenced field value in the cross-referenced segment.

To use this feature, create the index with the INDEX parameter set to NEW (the binary tree scheme). To determine the type of index used by a data source, enter the FDT command.
Syntax: How to Use the Extended LOOKUP Function

```plaintext
COMPUTE LOOKUP(field action);
```

where:

- **field**
  - Is the name of the field in the cross-referenced data source, used in a MODIFY computation. If the field name also exists in the host data source, you must qualify it here.

- **action**
  - Specifies the action the request takes. Valid values are:
    - **EQ** causes LOOKUP to take no further action if an exact match is not found. If a match is found, the value of rcode is set to 1; otherwise, it is set to 0. This is the default.
    - **GE** causes LOOKUP to locate the instance with the next highest value of the cross-referenced field. The value of rcode is set to 2.
    - **LE** causes LOOKUP to locate the instance with the next lowest value of the cross-referenced field. The value of rcode is set to -2.

Do not include a space between LOOKUP and the left parenthesis.

The following table shows the value of `rcode`, depending on which instance LOOKUP locates:

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exact cross-referenced value located.</td>
</tr>
<tr>
<td>2</td>
<td>Next highest cross-referenced value located.</td>
</tr>
<tr>
<td>-2</td>
<td>Next lowest cross-referenced value located.</td>
</tr>
<tr>
<td>0</td>
<td>Cross-referenced value not located.</td>
</tr>
</tbody>
</table>
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.

### Topics:

- **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**
- **DPART: Extracting a Component From a Date**
- **FIYR: Obtaining the Financial Year**
- **FIQTR: Obtaining the Financial Quarter**
- **FIYYQ: Converting a Calendar Date to a Financial Date**
- **TODAY: Returning the Current Date**
- **Using Legacy Date Functions**
- **AYM: Adding or Subtracting Months**
- **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**
- **DPART: Extracting a Component From a Date**
- **FIYR: Obtaining the Financial Year**
- **FIQTR: Obtaining the Financial Quarter**
- **FIYYQ: Converting a Calendar Date to a Financial Date**
- **GREGDT: Converting From Julian to Gregorian Format**
- **JULDAT: Converting From Gregorian to Julian Format**
- **YM: Calculating Elapsed Months**
Overview of Date Functions

The following explains the difference between the types of date functions:

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

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

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

In addition to the functions discussed in this topic, there are date and time functions that are available only in the Maintain language. For information on these functions, see *Maintain-specific Date and Time Functions* on page 305.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see *Accessing and Calling a Function* on page 43.
Using Standard Date Functions

In this section:
Specifying Work Days
Enabling Leading Zeros For Date and Time Functions in Dialogue Manager

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 202.

- 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 204.

For detailed information on each standard date function, see:

DATEADD: Adding or Subtracting a Date Unit to or From a Date on page 206
DATECVT: Converting the Format of a Date on page 209
DATEDIF: Finding the Difference Between Two Dates on page 211
DATEMOV: Moving a Date to a Significant Point on page 214
DATETRAN: Formatting Dates in International Formats on page 217
DPART: Extracting a Component From a Date on page 232
FIYR: Obtaining the Financial Year on page 234
FIQTR: Obtaining the Financial Quarter on page 236
FIYYQ: Converting a Calendar Date to a Financial Date on page 239
TODAY: Returning the Current Date on page 242
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

How to:

- Set Business Days
- View the Current Setting of Business Days

Business days are traditionally Monday through Friday, but not every business has this schedule. For example, if your company does business on Sunday, Tuesday, Wednesday, Friday, and Saturday, you can tailor business day units to reflect that schedule.

Syntax: How to Set Business Days

```
SET BUSDAYS = smtwtfs
```

where:

- `smtwtfs`

  Is the seven character list of days that represents your business week. The list has a position for each day from Sunday to Saturday:

  - To identify a day of the week as a business day, enter the first letter of that day in that day’s position.
  - To identify a non-business day, enter an underscore (_) in that day’s position.

If a letter is not in its correct position, or if you replace a letter with a character other than an underscore, you receive an error message.

Example: Setting Business Days to Reflect Your Work Week

The following designates work days as Sunday, Tuesday, Wednesday, Friday, and Saturday:

```
SET BUSDAYS = S_TW_FS
```
**Syntax:**

How to View the Current Setting of Business Days

```
? SET BUSDAYS
```

**Specifying Holidays**

**How to:**

Create a Holiday File

Select a Holiday File

**Reference:**

Rules for Creating a Holiday File

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. Calling a date function with a date value outside the range of the holiday file returns a zero for business day requests.
- You may include an optional description of the holiday, separated from the date by a space.

**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 203.
2. Save the file:

   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

This request uses HDAYTEST in its calculations:

SET BUSDAYS = SMTWTFS
SET HDAY = TEST
TABLE FILE MOVIES
PRINT TITLE RELDATE
COMPUTE NEXTDATE/YMD = DATEADD(RELDATE, 'BD', 1);
WHERE RELDATE GE '19910101';
END

Enabling Leading Zeros For Date and Time Functions in Dialogue Manager

**How to:**

Set the Display of Leading Zeros

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

The DATEADD function adds a unit to or subtracts a unit from a full component date format. A unit is one of the following:

- **Year.**
- **Month.** If the calculation using the month unit creates an invalid date, DATEADD corrects it to the last day of the month. For example, adding one month to October 31 yields November 30, not November 31 since November has 30 days.
- **Day.**
- **Weekday.** When using the weekday unit, DATEADD does not count Saturday or Sunday. For example, if you add one day to Friday, first DATEADD moves to the next weekday, Monday, then it adds a day. The result is Tuesday.
- **Business day.** When using the business day unit, DATEADD uses the BUSDAYS parameter setting and holiday file to determine which days are working days and disregards the rest. If Monday is not a working day, then one business day past Sunday is Tuesday. See *Specifying Holidays* on page 203 for more information.

DATEADD requires a date to be in date format. Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEADD requires a standard date stored as an offset from the base date, do not use DATEADD with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command* on page 53.

You add or subtract non day-based dates (for example, YM or YQ) directly without using DATEADD.

DATEADD works only with full component dates.
### Syntax: How to Add or Subtract a Date Unit to or From a Date

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

where:

- **date**
  - Date
  - Is a full component date.

- **component**
  - Alphanumeric
  - Is one of the following enclosed in single quotation marks:
    - **Y** indicates a year component.
    - **M** indicates a month component.
    - **D** indicates a day component.
    - **WD** indicates a weekday component.
    - **BD** indicates a business day component.

- **increment**
  - Integer
  - Is the number of date units added to or subtracted from `date`. If this number is not a whole unit, it is rounded down to the next largest integer.

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

### Example: Truncation With `DATEADD`

The number of units passed to `DATEADD` is always a whole unit. For example

```
DATEADD(DATE, 'M', 1.999)
```

adds one month because the number of units is less than two.

### Example: Using the Weekday Unit

If you use the weekday unit and a Saturday or Sunday is the input date, `DATEADD` changes the input date to Monday. The function

```
DATEADD('910623', 'WD', 1)
```
DATEADD: Adding or Subtracting a Date Unit to or From a Date

in which DATE is either Saturday or Sunday yields Tuesday; Saturday and Sunday are not weekdays, so DATEADD begins with Monday and adds one.

Note that the single quotes around the number in the first argument, ‘910623’, causes it to be treated as a natural date literal.

Example: Adding Weekdays to a Date (Reporting)

DATEADD adds three weekdays to NEW_DATE. In some cases, it adds more than three days because HIRE_DATE_PLUS_THREE would otherwise be on a weekend.

TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND HIRE_DATE AND COMPUTE
NEW_DATE/YYMD = HIRE_DATE;
HIRE_DATE_PLUS_THREE/YYMD = DATEADD(NEW_DATE, 'WD', 3);
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_DATE</th>
<th>HIRE_DATE_PLUS_THREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1982/04/01</td>
<td>1982/04/06</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1981/11/02</td>
<td>1981/11/05</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1982/04/01</td>
<td>1982/04/06</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1982/05/01</td>
<td>1982/05/06</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1981/07/01</td>
<td>1981/07/06</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1981/07/01</td>
<td>1981/07/06</td>
</tr>
</tbody>
</table>

Example: Determining If a Date Is a Work Day (Reporting)

DATEADD determines which values in the TRANSDATE field do not represent work days by adding zero days to TRANSDATE using the business day unit. If TRANSDATE does not represent a business day, DATEADD returns the next business day to DATEX. TRANSDATE is then compared to DATEX, and the day of the week is printed for all dates that do not match between the two fields, resulting in a list of all non-work days.

DEFINE FILE VIDEOTRK
DATEX/YMD  = DATEADD(TRANSDATE, 'BD', 0);
DATEINT/I8YYMD = DATECVT(TRANSDATE, 'YMD','I8YYMD');
END
TABLE FILE VIDEOTRK
SUM TRANSDATE NOPRINT
COMPUTE DAYNAME/A8 = DOWKL(DATEINT, DAYNAME); AS 'Day of Week'
BY TRANSDATE AS 'Date'
WHERE TRANSDATE NE DATEX
END
The output is:

<table>
<thead>
<tr>
<th>Date</th>
<th>Day of Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/06/22</td>
<td>SATURDAY</td>
</tr>
<tr>
<td>91/06/23</td>
<td>SUNDAY</td>
</tr>
<tr>
<td>91/06/30</td>
<td>SUNDAY</td>
</tr>
</tbody>
</table>

**Example:**  **Adding Months to a Date (Maintain)**

DATEADD adds months to the DATE1 field:

```
MAINTAIN
compute DATE1/yyyy = '20000101'
compute DATE2/yyyy = dateadd(date1, 'M', 2, date2);
type "DATE1 = <<DATE1 + 2 MONTHS = DATE2 = <<DATE2"
END
```

The result is:

```
DATE1 = 2000/01/01+ 2 MONTHS = DATE2 = 2000/03/01
```

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

### How to:
**Convert a Date Format**

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

DATECVT turns off optimization and compilation.

**Note:** You can use simple assignment instead of calling this function.

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

```
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.
DATECVT: Converting the Format of a Date

**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 enclosed in single quotation marks or a field containing the format. It is one of the following:

- A non-legacy date format (for example, YYMD, YQ, M, DMY, JUL).
- A legacy date format (for example, I6YMD or A8MDYY).
- A non-date format (such as I8 or A6). This format type causes DATECVT to convert the date into a full component date and return it as a whole number in the format provided.

**Example:** Converting a YYMD Date to DMY

DATECVT converts 19991231 to 311299 and stores the result in CONV_FIELD:

```
CONV_FIELD/DMY = DATECVT(19991231, 'I8YYMD', 'DMY');
```

or

```
CONV_FIELD/DMY = DATECVT('19991231', 'A8YYMD', 'DMY');
```

**Example:** Converting a Legacy Date to Date Format (Reporting)

DATECVT converts HIRE_DATE from I6YMD legacy date format to YYMD date format:

```
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND HIRE_DATE AND COMPUTE
NEW_HIRE_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD');
BY LAST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1982/04/01</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1981/11/02</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1982/04/01</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1982/05/01</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1981/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1981/07/01</td>
</tr>
</tbody>
</table>

**DATEDIF: Finding the Difference Between Two Dates**

**How to:**

Find 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 203 for more information.

DATEDIF returns a whole number. If the difference between two dates is not a whole number, DATEDIF truncates the value to the next largest integer. For example, the number of years between March 2, 2001, and March 1, 2002, is zero. If the end date is before the start date, DATEDIF returns a negative number.
You can find the difference between non-day based dates (for example YM or YQ) directly without using DATEDIF.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEDIF requires a standard date stored as an offset from the base date, do not use DATEDIF with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command* on page 53

DATEDIF works only with full component dates.

**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: Truncation With DATEDIF**

DATEDIF calculates the difference between March 2, 1996, and March 1, 1997, and returns a zero because the difference is less than a year:

```
DATEDIF('19960302', '19970301', 'Y')
```

**Example: Using Month Calculations**

The following expressions return a result of minus one month:

```
DATEDIF('19990228', '19990128', 'M')
DATEDIF('19990228', '19990129', 'M')
DATEDIF('19990228', '19990130', 'M')
DATEDIF('19990228', '19990131', 'M')
```

Additional examples:

```
DATEDIF('March 31 2001', 'May 31 2001', 'M') yields 2.
DATEDIF('March 31 2001', 'May 30 2001', 'M') yields 1 (because May 30 is not the end of the month).
DATEDIF('March 31 2001', 'April 30 2001', 'M') yields 1.
```

**Example: Finding the Number of Weekdays Between Two Dates (Reporting)**

DATECVT converts the legacy dates in HIRE_DATE and DAT_INC to the date format YYMD. DATEDIF then uses those date formats to determine the number of weekdays between NEW_HIRE_DATE and NEW_DAT_INC:

```sql
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND
COMPUTE NEW_HIRE_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD'); AND
COMPUTE NEW_DAT_INC/YYMD = DATECVT(DAT_INC, 'I6YMD', 'YYMD'); AND
COMPUTE WDAYS_HIRED/I8 = DATEDIF(NEW_HIRE_DATE, NEW_DAT_INC, 'WD');
BY LAST_NAME
IF WDAYS_HIRED NE 0
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEW_HIRE_DATE</th>
<th>NEW_DAT_INC</th>
<th>WDAYS_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>1982/01/04</td>
<td>1982/05/14</td>
<td>94</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>1982/02/02</td>
<td>1982/05/14</td>
<td>73</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>1982/01/04</td>
<td>1982/05/14</td>
<td>94</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>1980/06/02</td>
<td>1982/01/01</td>
<td>414</td>
</tr>
<tr>
<td></td>
<td>ALFRED</td>
<td>1980/06/02</td>
<td>1981/01/01</td>
<td>153</td>
</tr>
</tbody>
</table>
Example: Finding the Number of Years Between Two Dates (Maintain)

DATEDIF determines the number of years between DATE2 and DATE1:

```
MAINTAIN
Case Top
compute DATE1/ymd = '20020717';
compute DATE2/ymd = '19880705';
compute DIFF/I3 = DATEDIF(DATE2, DATE1, 'Y', DIFF);
type "<<DATE1 - <<DATE2 = <DIFF YEARS"
ENDCASE
END
```

The result is:

2002/07/17 - 1988/07/05 = 14 YEARS

DATEMOV: Moving a Date to a Significant Point

How to:
Move 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 more information, see Calling a Function From a Dialogue Manager Command on page 53

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

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

**Note:** DATEDMOV 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: Determining Significant Points for a Date (Reporting)

The BUSDAYS parameter sets the business days to Monday, Tuesday, Wednesday, and Thursday. DATECVT converts the legacy date HIRE_DATE to the date format YYMD and provides date display options. DATEMOV then determines significant points for HIRE_DATE.

```
SET BUSDAY = _MTWT_
TABLE FILE EMPLOYEE
PRINT
COMPUTE NEW_DATE/YYMD = DATECVT(HIRE_DATE, 'I6YMD', 'YYMD'); AND
COMPUTE NEW_DATE/WT = DATECVT(HIRE_DATE, 'I6YMD', 'WT'); AS 'DOW' AND
COMPUTE NWD/WT = DATEMOV(NEW_DATE, 'NWD'); AND
COMPUTE PWD/WT = DATEMOV(NEW_DATE, 'PWD'); AND
COMPUTE WDP/WT = DATEMOV(NEW_DATE, 'WD+'); AS 'WD+' AND
COMPUTE WDM/WT = DATEMOV(NEW_DATE, 'WD-'); AS 'WD-' AND
COMPUTE NBD/WT = DATEMOV(NEW_DATE, 'NBD'); AND
COMPUTE PBD/WT = DATEMOV(NEW_DATE, 'PBD'); AND
COMPUTE WBP/WT = DATEMOV(NEW_DATE, 'BD+'); AS 'BD+' AND
COMPUTE WBM/WT = DATEMOV(NEW_DATE, 'BD-'); AS 'BD-' BY LAST_NAME NOPRINT
HEADING
"Examples of DATEMOV"
"Business days are Monday, Tuesday, Wednesday, + Thursday"
""
"START DATE.. | MOVE POINTS........................."
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

```
Examples of DATEMOV
Business days are Monday, Tuesday, Wednesday, + Thursday

START DATE.. | MOVE POINTS.........................
NEW_DATE    DOW  NWD  PWD  WD+  WD-  NBD  PBD  BD+  BD-
--------    ---  ---  ---  ---  ---  ---  ---  ---  ---
1982/04/01  THU  WED  THU  THU  MON  WED  THU  THU
1981/11/02  MON  FRI  MON  MON  TUE  THU  MON  THU
1982/04/01  THU  WED  THU  THU  MON  WED  THU  THU
1982/05/01  SAT  TUE  THU  MON  FRI  TUE  WED  MON  THU
1981/07/01  WED  THU  TUE  WED  WED  THU  TUE  WED  WED
1981/07/01  WED  THU  TUE  WED  WED  THU  TUE  WED  WED
```
**Example:** Determining the End of the Week (Reporting)

DATEMOV determines the end of the week for each date in NEW_DATE and stores the result in EOW:

```
TABLE FILE EMPLOYEE
PRINT FIRST_NAME AND
COMPUTE NEW_DATE/YYMDWT = DATECVT(HIRE_DATE, 'I6YMD', 'YYMDWT'); AND
COMPUTE EOW/YYMDWT = DATEMOV(NEW_DATE, 'EOW');
BY LAST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEW_DATE</th>
<th>EOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>1982 AUG 1, SUN</td>
<td>1982 AUG 6, FRI</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>1982 JAN 4, MON</td>
<td>1982 JAN 8, FRI</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>1982 FEB 2, TUE</td>
<td>1982 FEB 5, FRI</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>1982 JUL 1, THU</td>
<td>1982 JUL 2, FRI</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>1982 JAN 4, MON</td>
<td>1982 JAN 8, FRI</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>1980 JUN 2, MON</td>
<td>1980 JUN 6, FRI</td>
</tr>
</tbody>
</table>

**Example:** Determining the End of the Week (Maintain)

DATEMOV determines the end of the week for each date:

```
MAINTAIN
COMPUTE X/YYMDWT='20020717';
COMPUTE Y/YYMDWT=DATEMOV(X, 'EOW', Y);
TYPE "<<X  "<<Y  END OF WEEK "
END
```

The result is:

```
2002/07/17, WED   2002/07/19, FRI END OF WEEK
```

**DATETRAN: Formatting Dates in International Formats**

**How to:**

Format Dates in International Formats

**Reference:**

Usage Notes for the DATETRAN Function

The DATETRAN function formats dates in international formats.
Syntax: How to Format Dates in International Formats

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

where:

indate

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

intype

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

These are the single component input types:

<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>

These are 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>

These are the three-component input types:
### Three-Component Input Type

| Description | Format Option |
|-------------|---------------|-------------|
| Four-digit year followed by month followed by day. | '(YYMD)' |
| Two-digit year followed by month followed by day. | '(YMD)' |
| Day component followed by month followed by four-digit year. | '(DMYY)' |
| Day component followed by month followed by two-digit year. | '(DMY)' |
| Month component followed by day followed by four-digit year. | '(MDYY)' |
| Month component followed by day followed by two-digit year. | '(MDY)' |
| Month component followed by day (derived from three-component date by ignoring year component). | '(MD)' |
| Day component followed by month (derived from three-component date by ignoring year component). | '(DM)' |

### 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 blank 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>
</tbody>
</table>
### DescriptionFormat Option

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

Valid month and day name translation options are:

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Displays month as an abbreviated name with no punctuation, all uppercase.</td>
</tr>
<tr>
<td>TR</td>
<td>Displays month as a full name, all uppercase.</td>
</tr>
<tr>
<td>Tp</td>
<td>Displays month as an abbreviated name followed by a period, all uppercase.</td>
</tr>
<tr>
<td>t</td>
<td>Displays month as an abbreviated name with no punctuation. The name is all lowercase or initial uppercase, depending on language code.</td>
</tr>
<tr>
<td>tr</td>
<td>Displays month as a full name. The name is all lowercase or initial uppercase, depending on language code.</td>
</tr>
<tr>
<td>tp</td>
<td>Displays month as an abbreviated name followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>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>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wp</td>
<td>Includes an abbreviated day of the week name at the start of the displayed date, all uppercase, followed by a period.</td>
</tr>
<tr>
<td>w</td>
<td>Includes an abbreviated day of the week name at the start of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>wr</td>
<td>Includes a full day of the week name at the start of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>wp</td>
<td>Includes an abbreviated day of the week name at the start of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).</td>
</tr>
<tr>
<td>x</td>
<td>Includes an abbreviated day of the week name at the end of the displayed date, all uppercase with no punctuation.</td>
</tr>
<tr>
<td>XR</td>
<td>Includes a full day of the week name at the end of the displayed date, all uppercase.</td>
</tr>
<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>xp</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>

Valid date delimiter options are:

<table>
<thead>
<tr>
<th>Format Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Uses a blank as the component delimiter. This is the default if the month or day of week is translated or if comma is used.</td>
</tr>
<tr>
<td>.</td>
<td>Uses a period as the component delimiter.</td>
</tr>
<tr>
<td>-</td>
<td>Uses a minus sign as the component delimiter. This is the default when the conditions for a blank default delimiter are not satisfied.</td>
</tr>
<tr>
<td>/</td>
<td>Uses a slash as the component delimiter.</td>
</tr>
<tr>
<td></td>
<td>Omits component delimiters.</td>
</tr>
<tr>
<td>K</td>
<td>Uses appropriate Asian characters as component delimiters.</td>
</tr>
<tr>
<td>c</td>
<td>Places a comma after the month name (following T, Tp, TR, t, tp, or tr). 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>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>Format Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</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

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

Reference: Usage Notes for the DATETRAN Function

- The output field, though it must be type A and not AnV, may in fact contain variable length information, since the lengths of month names and day names can vary, and also month and day numbers may be either one or two bytes long if a zero-suppression option is chosen. Unused bytes are filled with blanks.

- All invalid and inconsistent inputs result in all blank output strings. Missing data also results in blank output.

- The base dates (1900-12-31 and 1900-12 or 1901-01) are treated as though the DATEDISPLAY setting were ON (that is, not automatically shown as blanks). To suppress the printing of base dates, which have an internal integer value of 0, test for 0 before calling DATETRAN. For example:

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

- Valid translated date components are contained in files named DTLNG{lng} where lng is a three-character code that specifies the language. These files must be accessible for each language into which you want to translate dates.

- If you use a terminal emulator program, it must be set to use a code page that can display the accent marks and characters in the translated dates. You may not be able to display dates translated into European and Asian characters at the same time. Similarly, if you want to print the translated dates, your printer must be capable of printing the required characters.

- The DATETRAN function is not supported in Dialogue Manager.
**Example:** Using the DATETRAN Function

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

```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 HOLD FORMAT HTMLON TABLE SET PAGE-NUM OFF
ON TABLE SET STYLE *
GRID=OFF, $
END

Using Functions 225
```
The output is:

**FORMAT wr**

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

<table>
<thead>
<tr>
<th>Tuesday</th>
<th>Monday</th>
</tr>
</thead>
<tbody>
<tr>
<td>martes</td>
<td>lunes</td>
</tr>
<tr>
<td>mardi</td>
<td>lundi</td>
</tr>
<tr>
<td>Dienstag</td>
<td>Montag</td>
</tr>
</tbody>
</table>

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

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

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

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

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

<table>
<thead>
<tr>
<th>FORMAT Btdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank-delimited (B)</td>
</tr>
<tr>
<td>Abbreviated month name, default case (t)</td>
</tr>
<tr>
<td>Zero-suppress day number, end with suffix (do)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 4th 2005</td>
</tr>
</tbody>
</table>

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

```plaintext
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;
DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;
OUT3A/A12=DATETRAN(DATEYYMD, '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
OUT3B/A12=DATETRAN(DATEYYMD2, '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Btdp"
"
"Blank-delimited (B)"
"Abbreviated month name, default case (t)"
"Zero-suppress day number, end with period (dp)"
"German"
"
SUM OUT3A AS '' OUT3B AS '' TRANSDATE NOPRINT ON
TABLE HOLD FORMAT HTMLON TABLE SET PAGE-NUM OFF
END
```
The following request prints a blank delimited date in French with a full day name at the beginning and a full month name, in lowercase (the default for French):

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;
DATEW/W=TRANS1 ;
DATEW2/W=TRANS2 ;
DATEYYMD/YYMDW=TRANS1 ;
DATEYYMD2/YYMDW=TRANS2 ;
OUT4A/A30 = DATETRAN(DATEYYMD, '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
OUT4B/A30 = DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
END
```

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

```
FORMAT Bwrtr

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

mardi 04 janvier 2005   mercredi 02 mars 2005
```

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

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;
DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;
OUT5A/A30=DATETRAN(DATEYYMD, '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
OUT5B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
END

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

```
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
```

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

```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;
DATEW/W=TRANS1;
DATEW2/W=TRANS2;
DATEYYMD/YYMDW=TRANS1;
DATEYYMD2/YYMDW=TRANS2;
OUT6A/A30=DATETRAN(DATEYYMD, '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
OUT6B/A30=DATETRAN(DATEYYMD2, '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
END

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

<table>
<thead>
<tr>
<th>Ktrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Japanese characters (K in conjunction with the language code JA)</td>
</tr>
<tr>
<td>Full month name at beginning of date, default case (tr)</td>
</tr>
<tr>
<td>Zero-suppress day number (d)</td>
</tr>
<tr>
<td>Japanese</td>
</tr>
</tbody>
</table>

| 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:

```plaintext
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;
DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;
OUT7A/A30=DATETRAN(DATEYYMD , '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
OUT7B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
END

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

Using Functions 231
DPART: Extracting a Component From a Date

How to:
Extract a Date Component and Return It in Integer Format

The DPART function extracts a specified component from a date field and returns it in numeric format.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DPART requires a standard date stored as an offset from the base date, do not use DPART with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see Calling a Function From a Dialogue Manager Command on page 53

Syntax: How to Extract a Date Component and Return It in Integer Format

DPART (datevalue, 'component', output)

where:

datevalue
  Date
  Is a full component date.

component
  Alphanumeric
Is the name of the component to be retrieved enclosed in single quotation marks. Valid values are:

For year: YEAR, YY
For month: MONTH, MM
For day: DAY, For day of month: DAY-OF-MONTHDD.
For quarter: QUARTER, QQ

output
Integer

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

**Example: Extracting Date Components in Integer Format**

The following request against the VIDEOTRK data source uses the DPART function to extract the year, month, and day component from the TRANSDATE field:

```plaintext
DEFINE FILE
  VIDEOTRK
  YEAR/I4 = DPART(TRANSDATE, 'YEAR', 'I11');
  MONTH/I4 = DPART(TRANSDATE, 'MM', 'I11');
  DAY/I4 = DPART(TRANSDATE, 'DAY', 'I11');
END

TABLE FILE VIDEOTRK
PRINT TRANSDATE YEAR MONTH DAY
BY LASTNAME BY FIRSTNAME
WHERE LASTNAME LT 'DIAZ'
END
```
The output is:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>TRANSDATE</th>
<th>YEAR</th>
<th>MONTH</th>
<th>DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDREWS</td>
<td>NATALIA</td>
<td>91/06/19</td>
<td>1991</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/18</td>
<td>1991</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>BAKER</td>
<td>MARIE</td>
<td>91/06/19</td>
<td>1991</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/17</td>
<td>1991</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>BERTAL</td>
<td>MARCIA</td>
<td>91/06/23</td>
<td>1991</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/18</td>
<td>1991</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>CHANG</td>
<td>ROBERT</td>
<td>91/06/28</td>
<td>1991</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/27</td>
<td>1991</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/26</td>
<td>1991</td>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>COLE</td>
<td>ALLISON</td>
<td>91/06/24</td>
<td>1991</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91/06/23</td>
<td>1991</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>CRUZ</td>
<td>IVY</td>
<td>91/06/27</td>
<td>1991</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td>DAVIS</td>
<td>JASON</td>
<td>91/06/24</td>
<td>1991</td>
<td>6</td>
<td>24</td>
</tr>
</tbody>
</table>

**FIYR: Obtaining the Financial Year**

**How to:**
Obtain the Financial Year

The FIYR function returns the financial year, also known as the fiscal year, corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYR requires a standard date stored as an offset from the base date, do not use FIYR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command* on page 53

**Syntax:**

How to Obtain the Financial Year

`FIYR(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)`

where:

`inputdate`

Date

Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.
If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

**lowcomponent**

Alphanumeric

Is one of the following:

- D if the date contains a D or JUL component.
- M if the date contains an M component, but no D component.
- Q if the date contains a Q component.

**startmonth**

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

**startday**

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

**yrnumbering**

Alphanumeric

Valid values are:

- **FYE** to specify the *Financial Year Ending* convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

- **FYS** to specify the *Financial Year Starting* convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

**output**

I, Y, or YY

The result will be in integer format, or Y or YY. This function returns a year value. In case of an error, zero is returned.
Note: February 29 cannot be used as a start day for a financial year.

Example: Obtaining the Financial Year

The following request against the CENTSTMT data source obtains the financial year corresponding to an account period (field PERIOD, format YYM) and returns the values in each of the supported formats: Y, YY, and I4.

```
DEFINE FILE CENTSTMT
FISCALYY/YY=FIYR(PERIOD,'M', 4,1,'FYE',FISCALYY);
FISCALY/Y=FIYR(PERIOD,'M', 4,1,'FYE',FISCALY);
FISCALI/I4=FIYR(PERIOD,'M', 4,1,'FYE',FISCALI);
END

TABLE FILE CENTSTMT
PRINT PERIOD FISCALYY FISCALY FISCALI
BY GL_ACCOUNT
WHERE GL_ACCOUNT LT '2100'
END
```

On the output, note that the period April 2002 (2002/04) is in fiscal year 2003 because the starting month is April (4), and the FYE numbering convention is used:

<table>
<thead>
<tr>
<th>Ledger Account</th>
<th>PERIOD</th>
<th>FISCALYY</th>
<th>FISCALY</th>
<th>FISCALI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2002/01</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/02</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/03</td>
<td>2002</td>
<td>03</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/04</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td>2002/05</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td>2002/06</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td>2000</td>
<td>2002/01</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/02</td>
<td>2002</td>
<td>02</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/03</td>
<td>2002</td>
<td>03</td>
<td>2002</td>
</tr>
<tr>
<td></td>
<td>2002/04</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td>2002/05</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
<tr>
<td></td>
<td>2002/06</td>
<td>2003</td>
<td>03</td>
<td>2003</td>
</tr>
</tbody>
</table>

FIQTR: Obtaining the Financial Quarter

How to:

Obtain the Financial Quarter

The FIQTR function returns the financial quarter corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.
Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIQTR requires a standard date stored as an offset from the base date, do not use FIQTR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see *Calling a Function From a Dialogue Manager Command* on page 53

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

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**

The following request against the CENTHR data source obtains the financial quarter corresponding to an employee starting date (field START_DATE, format YYMD) and returns the values in each of the supported formats: Q and I1.

```plaintext
DEFINE FILE CENTHR
FISCALQ/Q=FIQTR(START_DATE,'D',10,1,'FYE',FISCALQ);
FISCALI/I1=FIQTR(START_DATE,'D',10,1,'FYE',FISCALI);
END

TABLE FILE CENTHR
PRINT START_DATE FISCALQ FISCALI
BY LNAME BY FName
WHERE LNAME LIKE 'C%'
END
```
On the output, note that the date November 12, 1998 (1998/11/12) is in fiscal quarter Q1 because the starting month is October (10):

<table>
<thead>
<tr>
<th>Last</th>
<th>First</th>
<th>Starting</th>
<th>FISCALQ</th>
<th>FISCALI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARNEY</td>
<td>ROSS</td>
<td>1998/09/12</td>
<td>Q4</td>
<td>4</td>
</tr>
<tr>
<td>CHIEN</td>
<td>CHRISTINE</td>
<td>1997/10/01</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>CLEVELAND</td>
<td>PHILIP</td>
<td>1996/07/30</td>
<td>Q4</td>
<td>4</td>
</tr>
<tr>
<td>CLINE</td>
<td>STEPHEN</td>
<td>1998/11/12</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>COHEN</td>
<td>DANIEL</td>
<td>1997/10/05</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>CORRIEUA</td>
<td>RAYMOND</td>
<td>1997/12/05</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>COSSMAN</td>
<td>MARK</td>
<td>1996/12/19</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>CRAWIN</td>
<td>CHRIS</td>
<td>1996/12/03</td>
<td>Q1</td>
<td>1</td>
</tr>
<tr>
<td>CROWDER</td>
<td>WESLEY</td>
<td>1996/09/17</td>
<td>Q4</td>
<td>4</td>
</tr>
<tr>
<td>CULLEN</td>
<td>DENNIS</td>
<td>1995/09/05</td>
<td>Q4</td>
<td>4</td>
</tr>
<tr>
<td>CUMMINGS</td>
<td>JAMES</td>
<td>1993/07/11</td>
<td>Q4</td>
<td>4</td>
</tr>
<tr>
<td>CUTLIP</td>
<td>GREGG</td>
<td>1997/03/26</td>
<td>Q2</td>
<td>2</td>
</tr>
</tbody>
</table>

FIYYQ: Converting a Calendar Date to a Financial Date

How to:

Convert a Calendar Date to a Financial Date

The FIYYQ function returns a financial date containing both the financial year and quarter that corresponds to a given calendar date. The returned financial date is based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYYQ requires a standard date stored as an offset from the base date, do not use FIYYQ with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

For more information, see Calling a Function From a Dialogue Manager Command on page 53

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

The following request against the CENTHR data source converts each employee starting date (field START_DATE, format YYMD) to a financial date containing year and quarter components in all the supported formats: YQ, YYQ, QY, and QYY.

DEFINE FILE CENTHR
FISYQ/YQ=FIYYQ(START_DATE,'D',10,1,'FYE',FISYQ);
FISYYQ/YYQ=FIYYQ(START_DATE,'D',10,1,'FYE',FISYYQ);
FISQY/QY=FIYYQ(START_DATE,'D',10,1,'FYE',FISQY);
FISQYY/QYY=FIYYQ(START_DATE,'D',10,1,'FYE',FISQYY);
END

TABLE FILE CENTHR
PRINT START_DATE FISYQ FISYYQ FISQY FISQYY
BY LNAME BY FNAME
WHERE LNAME LIKE 'C%'
END

On the output, note that the date November 12, 1998 (1998/11/12) is converted to Q1 1999 because the starting month is October (10), and the FYE numbering convention is used:

<table>
<thead>
<tr>
<th>Last Name</th>
<th>First</th>
<th>Starting Date</th>
<th>FISYQ</th>
<th>FISYYQ</th>
<th>FISQY</th>
<th>FISQYY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIEN</td>
<td>CHRISTINE</td>
<td>1997/10/01</td>
<td>98</td>
<td>Q1</td>
<td>1998</td>
<td>Q1 98</td>
</tr>
<tr>
<td>CLEVELAND</td>
<td>PHILIP</td>
<td>1996/07/30</td>
<td>96</td>
<td>Q4</td>
<td>1996</td>
<td>Q4 96</td>
</tr>
<tr>
<td>CLINE</td>
<td>STEPHEN</td>
<td>1998/11/12</td>
<td>99</td>
<td>Q1</td>
<td>1999</td>
<td>Q1 99</td>
</tr>
<tr>
<td>COHEN</td>
<td>DANIEL</td>
<td>1997/10/05</td>
<td>98</td>
<td>Q1</td>
<td>1998</td>
<td>Q1 98</td>
</tr>
<tr>
<td>CORRIEVAU</td>
<td>RAYMOND</td>
<td>1997/12/05</td>
<td>98</td>
<td>Q1</td>
<td>1998</td>
<td>Q1 98</td>
</tr>
<tr>
<td>COSSMAN</td>
<td>MARK</td>
<td>1996/12/19</td>
<td>97</td>
<td>Q1</td>
<td>1997</td>
<td>Q1 97</td>
</tr>
<tr>
<td>CRONIN</td>
<td>CHRIS</td>
<td>1996/12/03</td>
<td>97</td>
<td>Q1</td>
<td>1997</td>
<td>Q1 97</td>
</tr>
<tr>
<td>CROWDER</td>
<td>WESLEY</td>
<td>1996/09/17</td>
<td>96</td>
<td>Q4</td>
<td>1996</td>
<td>Q4 96</td>
</tr>
<tr>
<td>CULLEN</td>
<td>DENNIS</td>
<td>1995/09/05</td>
<td>95</td>
<td>Q4</td>
<td>1995</td>
<td>Q4 95</td>
</tr>
<tr>
<td>CUMMINGS</td>
<td>JAMES</td>
<td>1993/07/11</td>
<td>93</td>
<td>Q4</td>
<td>1993</td>
<td>Q4 93</td>
</tr>
<tr>
<td>CUTLIP</td>
<td>GREGG</td>
<td>1997/03/26</td>
<td>97</td>
<td>Q2</td>
<td>1997</td>
<td>Q2 97</td>
</tr>
</tbody>
</table>
TODAY: Returning the Current Date

**How to:**
Retrieve 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.

A compiled MODIFY procedure must use TODAY to obtain the date. It cannot use the system variables.

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

TODAY(output)

where:

output

Alphanumeric, at least A8

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

The following apply:

- If 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 the DATE field. The request then displays the date in the page heading.

```plaintext
DEFINE FILE EMPLOYEE
DATE/A10 WITH EMP_ID = TODAY(DATE);
END
```
The legacy date functions were created for use with dates in integer, packed decimal, or alphanumeric format.

For detailed information on each legacy date function, see:

- **AYM**: Adding or Subtracting Months on page 246
- **AYMD**: Adding or Subtracting Days on page 247
- **CHGDAT**: Changing How a Date String Displays on page 249
- **DA Functions**: Converting a Legacy Date to an Integer on page 252
- **DMY, MDY, YMD**: Calculating the Difference Between Two Dates on page 253
- **DOWK and DOWKL**: Finding the Day of the Week on page 255
- **DT Functions**: Converting an Integer to a Date on page 256
- **GREGDT**: Converting From Julian to Gregorian Format on page 257
- **JULDAT**: Converting From Gregorian to Julian Format on page 259
- **YM**: Calculating Elapsed Months on page 261
Using Old Versions of Legacy Date Functions

How to: Activate Old 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.

Using Dates With Two- and Four-Digit Years

Legacy date functions accept dates with two- or four-digit years. Four-digit years that display the century, such as 2000 or 1900, can be used if their formats are specified as I8YYMD, P8YYMD, D8YYMD, F8YYMD, or A8YYMD. Two-digit years can use the DEFCENT and YRTHRESH parameters to assign century values if the field has a length of six (for example, I6YMD). For information on these parameters, see Customizing Your Environment in Developing Applications.
**Example: Using Four-Digit Years**

The EDIT function creates dates with four-digit years. The functions JULDAT and GREGDT then convert these dates to Julian and Gregorian formats.

```plaintext
DEFINE FILE EMPLOYEE
DATE/I8YYMD = EDIT('19'|EDIT(HIRE_DATE));
JDATE/I7 = JULDAT(DATE, 'I7');
GDATE/I8 = GREGDT(JDATE, 'I8');
END

TABLE FILE EMPLOYEE
PRINT DATE JDATE GDATE
END

The output is:

<table>
<thead>
<tr>
<th>DATE</th>
<th>JDATE</th>
<th>GDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980/06/02</td>
<td>1980154</td>
<td>19800602</td>
</tr>
<tr>
<td>1981/07/01</td>
<td>1981182</td>
<td>19810701</td>
</tr>
<tr>
<td>1982/05/01</td>
<td>1982121</td>
<td>19820501</td>
</tr>
<tr>
<td>1982/01/04</td>
<td>1982004</td>
<td>19820104</td>
</tr>
<tr>
<td>1982/08/01</td>
<td>1982213</td>
<td>19820801</td>
</tr>
<tr>
<td>1982/01/04</td>
<td>1982004</td>
<td>19820104</td>
</tr>
<tr>
<td>1982/07/01</td>
<td>1982182</td>
<td>19820701</td>
</tr>
<tr>
<td>1981/07/01</td>
<td>1981182</td>
<td>19810701</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1982/02/02</td>
<td>1982033</td>
<td>19820202</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1981/11/02</td>
<td>1981306</td>
<td>19811102</td>
</tr>
<tr>
<td>1982/04/01</td>
<td>1982091</td>
<td>19820401</td>
</tr>
<tr>
<td>1982/05/15</td>
<td>1982135</td>
<td>19820515</td>
</tr>
</tbody>
</table>
```

**Example: Using Two-Digit Years**

The AYMD function returns an eight-digit date when the input argument has a six-digit legacy date format. Since DEFCENT is 19 and YRTHRESH is 83, year values from 83 through 99 are interpreted as 1983 through 1999, and year values from 00 through 82 are interpreted as 2000 through 2082.

```plaintext
SET DEFCENT=19, YRTHRESH=83

DEFINE FILE EMPLOYEE
NEW_DATE/I8YYMD = AYMD(EFFECT_DATE, 30, 'I8');
END

TABLE FILE EMPLOYEE
PRINT EFFECT_DATE NEW_DATE BY EMP_ID
END
```
### AYM: Adding or Subtracting Months

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>EFFECT_DATE</th>
<th>NEW_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>82/11/01</td>
<td>2082/12/01</td>
</tr>
<tr>
<td>112847612</td>
<td>83/01/01</td>
<td>1983/01/31</td>
</tr>
<tr>
<td>117593129</td>
<td>83/03/01</td>
<td>1983/03/31</td>
</tr>
<tr>
<td>119265415</td>
<td>84/09/01</td>
<td>1984/10/01</td>
</tr>
<tr>
<td>219984371</td>
<td>82/12/01</td>
<td>2082/12/31</td>
</tr>
<tr>
<td>326179357</td>
<td>83/05/01</td>
<td>1983/05/31</td>
</tr>
<tr>
<td>451123478</td>
<td>84/01/01</td>
<td>1984/01/01</td>
</tr>
<tr>
<td>543729165</td>
<td>85/01/01</td>
<td>1985/01/01</td>
</tr>
</tbody>
</table>

### How to:

Add or Subtract Months to or From a Date

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

### Syntax:

**How to Add or Subtract Months to or From a Date**

\[
AYM(\text{indate}, \text{months}, \text{output})
\]

where:

- **indate**
  - I4, I4YM, I6, or I6YYM
  - Is the legacy date in year-month format, the name of a field that contains the date, or an expression that returns the date. If the date is not valid, the function returns the value 0 (zero).

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

I4YM or I6YYM

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

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

Example: Adding Months to a Date

The COMPUTE command converts the dates in HIRE_DATE from year-month-day to year-month format and stores the result in HIRE_MONTH. AYM then adds six months to HIRE_MONTH and stores the result in AFTER6MONTHS:

TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
HIRE_MONTH/I4YM = HIRE_DATE/100 ;
AFTER6MONTHS/I4YM = AYM(HIRE_MONTH, 6, AFTER6MONTHS) ;
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>HIRE_MONTH</th>
<th>AFTER6MONTHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>82/04</td>
<td>82/10</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>81/11</td>
<td>82/05</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>82/04</td>
<td>82/10</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>82/05</td>
<td>82/11</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>81/07</td>
<td>82/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>81/07</td>
<td>82/01</td>
</tr>
</tbody>
</table>

AYMD: Adding or Subtracting Days

How to:

Add or Subtract Days to or From a Date

The AYMD function adds days to or subtracts days from a date in year-month-day format. You can convert a date to this format using the CHGDAT or EDIT function.
**Syntax:** How to Add or Subtract Days to or From a Date

\[ \text{AYMD}(\text{indate}, \text{days}, \text{output}) \]

where:

**indate**

\( \text{I6, I6YMD, I8, I8YYMD} \)

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 *indate*. To subtract days, use a negative number.

**output**

\( \text{I6, I6YMD, I8, or I8YYMD} \)

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. If *indate* is a field, *output* must have the same format.

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

**Example:** Adding Days to a Date

AYMD adds 35 days to each value in the HIRE_DATE field, and stores the result in AFTER35DAYS:

```plaintext
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
AFTER35DAYS/I6YMD = \text{AYMD}(\text{HIRE_DATE}, \text{35}, \text{AFTER35DAYS});
BY LAST_NAME BY FIRST_NAME
WHERE \text{DEPARTMENT} \text{ EQ} '\text{PRODUCTION}';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>AFTER35DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>82/09/05</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>82/02/08</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>82/03/09</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>82/08/05</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>82/02/08</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>80/07/07</td>
</tr>
</tbody>
</table>
CHGDAT: Changing How a Date String Displays

How to:
Change the Date Display String

Reference:
Short to Long Conversion

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

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

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

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day of the month (01 through 31).</td>
</tr>
<tr>
<td>M</td>
<td>Month of the year (01 through 12).</td>
</tr>
<tr>
<td>Y[Y]</td>
<td>Year. Y indicates a two-digit year (such as 94); YY indicates a four-digit year (such as 1994).</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Displays the month as a three-letter abbreviation.</td>
</tr>
<tr>
<td>X</td>
<td>Displays the full name of the month.</td>
</tr>
</tbody>
</table>
Display options can consist of up to five display characters. Characters other than those display options are ignored.

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

**Note:** Display options are *not* date formats.

**Reference:** **Short to Long Conversion**

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

<table>
<thead>
<tr>
<th>Portion of Date Missing</th>
<th>Portion Supplied by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day (for example, from YM to YMD)</td>
<td>Last day of the month.</td>
</tr>
<tr>
<td>Month (for example, from Y to YM)</td>
<td>Last month of the year (December).</td>
</tr>
<tr>
<td>Year (for example, from MD to YMD)</td>
<td>The year 99.</td>
</tr>
<tr>
<td>Converting year from two-digit to four-digit (for example, from YMD to YYMD)</td>
<td>If DATEFNS=ON, the century will be determined by the 100-year window defined by DEFCENT and YRTHRESH. See <em>Working With Cross-Century Dates</em> in <em>Developing Applications</em> for details on DEFCENT and YRTHRESH. If DATEFNS=OFF, the year 19xx is supplied, where xx is the last two digits in the year.</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.

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

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

Example: Converting the Date Display From YMD to MDYYX

The EDIT function changes HIRE_DATE from numeric to alphanumeric format. CHGDAT then converts each value in ALPHA_HIRE from displaying the components as YMD to MDYYX and stores the result in HIRE_MDY, which has the format A17. The option X in the output value displays the full name of the month.

TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
ALPHA_HIRE/A17 = EDIT(HIRE_DATE); NOPRINT AND COMPUTE
HIRE_MDY/A17 = CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>HIRE_MDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>AUGUST 01 1982</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>JANUARY 04 1982</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>FEBRUARY 02 1982</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>JULY 01 1982</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>JANUARY 04 1982</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>JUNE 02 1980</td>
</tr>
</tbody>
</table>

**DA Functions: Converting a Legacy Date to an Integer**

**How to:**

Convert a Date to an Integer

The DA functions convert a legacy date to the number of days between it and a base date (December 31, 1899). By converting a date to the number of days, you can add and subtract dates and calculate the intervals between them, or you can add to or subtract numbers from the dates to get new dates.

You can convert the result back to a date using the DT functions discussed in *DT Functions: Converting an Integer to a Date* on page 256.

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, or the name of a field that contains the date. The date is truncated to an integer before conversion. If \textit{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.

\textbf{output}

\textbf{Integer}

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The format of the date returned depends on the function.

\textbf{Example: Converting Dates and Calculating the Difference Between Them}

DAYMD converts the DAT\_INC and HIRE\_DATE fields to the number of days since December 31, 1899, and the smaller number is then subtracted from the larger number:

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT DAT\_INC AS 'RAISE DATE' AND COMPUTE
DAYS\_HIRED/I8 = DAYMD(DAT\_INC, 'I8') - DAYMD(HIRE\_DATE, 'I8');
BY LAST\_NAME BY FIRST\_NAME
IF DAYS\_HIRED NE 0
WHERE DEPARTMENT EQ 'PRODUCTION';
END
\end{verbatim}

The output is:

\begin{verbatim}
     LAST\_NAME     FIRST\_NAME  RAISE DATE  DAYS\_HIRED
------------     ----------  ----------  ----------
    IRVING        JOAN          82/05/14         130
    MCKNIGHT      ROGER         82/05/14         101
     SMITH        RICHARD       82/05/14         130
       STEVENS     ALFRED        82/01/01         578
           --------          --------
       81/01/01         213
\end{verbatim}

\textbf{DMY, MDY, YMD: Calculating the Difference Between Two Dates}

\textbf{How to:}

Calculate 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

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

where:

- **function**
  - Is one of the following:
    - **DMY** calculates the difference between two dates in day-month-year format.
    - **MDY** calculates the difference between two dates in month-day-year format.
    - **YMD** calculates the difference between two dates in year-month-day format.

- **from_date**
  - I, P, or A format with date display options.
  - Is the beginning legacy date, or the name of a field that contains the date.

- **to_date**
  - I, P, or A format with date display options.I6xxx or I8xxx where xxx corresponds to the specified function (DMY, YMD, or MDY).
  - Is the end date, or the name of a field that contains the date.

**Example:**

Calculating the Number of Days Between Two Dates

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

```
TABLE FILE EMPLOYEE
SUM HIRE_DATE FST.DAT_INC AS 'FIRST PAY, INCREASE' AND COMPUTE
DIFF/I4 = YMD(HIRE_DATE, FST.DAT_INC) AS 'DAYS, BETWEEN'
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>INCREASE</th>
<th>DAYS BETWEEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>82/04/01</td>
<td>0</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>82/04/09</td>
<td>158</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>82/06/11</td>
<td>71</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>82/06/01</td>
<td>31</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
</tr>
</tbody>
</table>
DOWK and DOWKL: Finding the Day of the Week

How to:
Find the Day of the Week

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

Syntax: How to Find the Day of the Week

\{DOWK|DOWKL\}(\textit{indate}, \textit{output})

where:

\textit{indate}

I6YMD or I8YMD

Is the legacy date in year-month-day format. If the date is not valid, the function returns spaces. If the date specifies a two digit year and DEFCENT and YRTHRESH values have not been set, the function assumes the 20th century.

\textit{output}

DOWK: A4. DOWKL: A12

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

Example: Finding the Day of the Week

DOWK determines the day of the week that corresponds to the value in the HIRE_DATE field and stores the result in DATED:

\begin{verbatim}
TABLE FILE EMPLOYEE
PRINT EMP_ID AND HIRE_DATE AND COMPUTE
DATED/A4 = DOWK(HIRE_DATE, DATED);
WHERE DEPARTMENT EQ 'PRODUCTION';
END
\end{verbatim}

The output is:

\begin{verbatim}
EMP_ID     HIRE_DATE  DATED
------     ---------  -----  
071382660  80/06/02  MON
119265415  82/01/04  MON
119329144  82/08/01  SUN
123764317  82/01/04  MON
126724188  82/07/01  THU
451123478  82/02/02  TUE
\end{verbatim}
How to: Convert an Integer to a Date

The DT functions convert an integer representing the number of days elapsed since December 31, 1899 to the corresponding date. They are useful when you are performing arithmetic on a date converted to the number of days (for more information, see DA Functions: Converting a Legacy Date to an Integer on page 252). The DT functions convert the result back to a date.

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

**Note:** When USERFNS is set to LOCAL, DT functions only display a six-digit date.

**Syntax:** How to Convert an Integer to a Date

```plaintext
function(number, output)
```

where:

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

- **number**
  - Integer
  - Is the number of days since December 31, 1899. The number is truncated to an integer.

- **output**
  - I8xxx, where xxx corresponds to the function DTxxx in the above list.
  - Is the name of the field containing the result or the format of the output value enclosed in single quotation marks. The output format depends on the function being used.
### Example: Converting an Integer to a Date

DTMDY converts the NEWF field (which was converted to the number of days by DAYMD) to the corresponding date and stores the result in NEW_HIRE_DATE:

```plaintext
-* THIS PROCEDURE CONVERTS HIRE_DATE, WHICH IS IN I6YMD FORMAT, 
-* TO A DATE IN I8MDYY FORMAT. 
-* FIRST IT USES THE DAYMD FUNCTION TO CONVERT HIRE_DATE 
-* TO A NUMBER OF DAYS. 
-* THEN IT USES THE DTMDY FUNCTION TO CONVERT THIS NUMBER OF 
-* DAYS TO I8MDYY FORMAT 
_*
```

```plaintext
DEFINE FILE EMPLOYEE
NEWF/I8 WITH EMP_ID = DAYMD(HIRE_DATE, NEWF);
NEW_HIRE_DATE/I8MDYY WITH EMP_ID = DTMDY(NEWF, NEW_HIRE_DATE);
END TABLE FILE EMPLOYEE
PRINT HIRE_DATE NEW_HIRE_DATE
BY FN BY LN
WHERE DEPARTMENT EQ 'MIS'
END
```

The output is:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>NEW_HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARBARA</td>
<td>CROSS</td>
<td>81/11/02</td>
<td>11/02/1981</td>
</tr>
<tr>
<td>DIANE</td>
<td>JONES</td>
<td>82/05/01</td>
<td>05/01/1982</td>
</tr>
<tr>
<td>JOHN</td>
<td>MCCOY</td>
<td>81/07/01</td>
<td>07/01/1981</td>
</tr>
<tr>
<td>MARY</td>
<td>GREENSPAN</td>
<td>82/04/01</td>
<td>04/01/1982</td>
</tr>
<tr>
<td></td>
<td>SMITH</td>
<td>81/07/01</td>
<td>07/01/1981</td>
</tr>
<tr>
<td>ROSEMARIE</td>
<td>BLACKWOOD</td>
<td>82/04/01</td>
<td>04/01/1982</td>
</tr>
</tbody>
</table>

### GREGDT: Converting From Julian to Gregorian Format

#### How to:
Convert From Julian to Gregorian Format

#### Reference:
DATEFNS Settings for GREGDT

The GREGDT function converts a date in Julian format (year-day) to Gregorian format (year-month-day).

A date in Julian format is a five- or seven-digit number. The first two or four digits are the year; the last three digits are the number of the day, counting from January 1. For example, January 1, 1999 in Julian format is either 99001 or 1999001; June 21, 2004 in Julian format is 2004173.
**Reference: DATEFNS Settings for GREGDT**

GREGDT converts a Julian date to either YMD or YYMD format using the DEFCENT and YRTHRESH parameter settings to determine the century, if required. GREGDT returns a date as follows:

<table>
<thead>
<tr>
<th>DATEFNS Setting</th>
<th>I6 or I7 Format</th>
<th>I8 Format or Greater</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>YMD</td>
<td>YYMD</td>
</tr>
<tr>
<td>OFF</td>
<td>YMD</td>
<td>YMD</td>
</tr>
</tbody>
</table>

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

GREGDT(indate, output)

where:

indate

I5 or I7

Is the Julian date, which is truncated to an integer before conversion. Each value must be a five- or seven-digit number after truncation. If the date is invalid, the function returns a 0 (zero).

output

I6, I8, I6YMD, or I8YYMD

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

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

GREGDT converts the JULIAN field to YYMD (Gregorian) format. It determines the century using the default DEFCENT and YRTHRESH parameter settings.

```
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND
COMPUTE JULIAN/I5 = JULDAT(HIRE_DATE, JULIAN); AND
COMPUTE GREG_DATE/I8 = GREGDT(JULIAN, 'I8');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>JULIAN</th>
<th>GREG_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>82213</td>
<td>19820801</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>82004</td>
<td>19820104</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>82033</td>
<td>19820202</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>82182</td>
<td>19820701</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>82004</td>
<td>19820104</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>80154</td>
<td>19800602</td>
</tr>
</tbody>
</table>

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

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

**Reference:**
DATEFNS Settings for JULDAT

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

**Reference:** DATEFNS Settings for JULDAT

JULDAT converts a Gregorian date to either YYNNN or YYYYYNNN format, using the DEFCENT and YRTHRESH parameter settings to determine if the century is required.

JULDAT returns dates as follows:

<table>
<thead>
<tr>
<th>DATEFNS Setting</th>
<th>I6 or I7 Format</th>
<th>I8 Format or Greater</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>YYNNN</td>
<td>YYYYYNNN</td>
</tr>
<tr>
<td>OFF</td>
<td>YYNNN</td>
<td>YYNNN</td>
</tr>
</tbody>
</table>
Syntax: How to Convert From Gregorian to Julian Format

JULDAT(indate, output)
where:

indate

I6, I8, I6YMD, I8YYMD
Is the legacy date to convert or the name of the field that contains the date in year-month-day format (YMD or YYMD).

output

I5 or I7
Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Converting From Gregorian to Julian Format

JULDAT converts the HIRE_DATE field to Julian format. It determines the century using the default DEFCENT and YRTHRESH parameter settings.

TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
JULIAN/I7 = JULDAT(HIRE_DATE, JULIAN);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>JULIAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>1982213</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>1982004</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>1982033</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>1982182</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>1982004</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>1980154</td>
</tr>
</tbody>
</table>
YM: Calculating Elapsed Months

How to: Calculate 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

YM(fromdate, todate, output)

where:

*fromdate*

I4YM or I6YYM

Is the start date in year-month format (for example, I4YM). If the date is not valid, the function returns the value 0 (zero).

*todate*

I4YM or I6YYM

Is the end date in year-month format. If the date is not valid, the function returns the value 0 (zero).

*output*

Integer

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

**Tip:** If fromdate or todate is in integer year-month-day format (I6YMD or I8YYMD), simply divide by 100 to convert to year-month format and set the result to an integer. This drops the day portion of the date, which is now after the decimal point.
**Example:**  **Calculating Elapsed Months**

The COMPUTE commands convert the dates from year-month-day to year-month format; then YM calculates the difference between the values in the HIRE_DATE/100 and DAT_INC/100 fields:

```
TABLE FILE EMPLOYEE
PRINT DAT_INC AS 'RAISE DATE' AND COMPUTE
HIRE_MONTH/I4YM = HIRE_DATE/100; NOPRINT AND COMPUTE
MONTH_INC/I4YM = DAT_INC/100; NOPRINT AND COMPUTE
MONTHS_HIRED/I3 = YM(HIRE_MONTH, MONTH_INC, 'I3');
BY LAST_NAME BY FIRST_NAME BY HIRE_DATE
IF MONTHS_HIRED NE 0
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>RAISE_DATE</th>
<th>MONTHS_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>82/04/09</td>
<td>5</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>82/06/11</td>
<td>2</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>82/06/01</td>
<td>1</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>6</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>6</td>
</tr>
</tbody>
</table>
```
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.

**Topics:**
- 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
- HEXTR: Extracting Components of a Date-Time Value and Setting Remaining Components to Zero
- HGETC: Storing the Current Date and Time in a Date-Time Field
- HHMMSS: Retrieving the Current Time
- HINPUT: Converting an Alphanumeric String to a Date-Time Value
- HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight
- HMASK: Extracting Components of a Date-Time Field and Preserving Remaining Components
- HNAME: Retrieving a Date-Time Component in Alphanumeric Format
- HPART: Retrieving a Date-Time Component as a Numeric Value
- HSETPT: Inserting a Component Into a Date-Time Value
- HTIME: Converting the Time Portion of a Date-Time Value to a Number
- HTMTOTS or TIMETOTS: Converting a Time to a Timestamp
- HYYWD: Returning the Year and Week Number From a Date-Time Value
Using Date-Time Functions

In this section:
Date-Time Parameters
Supplying Arguments for Date-Time Functions
Using Date-Time Formats
Assigning Date-Time Values

The functions described in this section operate on fields in date-time format (sometimes called H format).

Date-Time Parameters

In this section:
Specifying the Order of Date Components
Specifying the First Day of the Week for Use in Date-Time Functions
Controlling Processing of Date-Time Values

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

How to:
Specify the Order of Date Components in a Date-Time Field

The DATEFORMAT parameter specifies the order of the date components (month/day/year) when date-time values are entered in the formatted string and translated string formats described in Using Date-Time Formats on page 269. It makes the input format of a value independent of the format of the variable to which it is being assigned.
Syntax: **How to Specify the Order of Date Components in a Date-Time Field**

SET DATEFORMAT = *option*

where:

*option*

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

Example: **Using the DATEFORMAT Parameter**

The following request uses a natural date literal with ambiguous numeric day and month components (APR 04 05) as input to the HINPUT function:

```plaintext
SET DATEFORMAT = MYD
DEFINE FILE EMPLOYEE
DTFLDYYMD/HYYMDI = HINPUT(9,'APR 04 05', 8, DTFLDYYMD);
END

TABLE FILE EMPLOYEE
SUM CURR_SAL NOPRINT DTFLDYYMD
END
```

With DATEFORMAT set to MYD, the value is interpreted as April 5, 1904:

```
DTFLDYYMD
---------
1904-04-05 00:00
```

Specifying the First Day of the Week for Use in Date-Time Functions

**How to:**

Set a Day as the Start of the Week

View the Current Setting of WEEKFIRST

The WEEKFIRST parameter specifies a day of the week as the start of the week. This is used in week computations by the HDIFF, HNAME, HPART, and HSETPT functions. 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, HYYWD, and HNAME subroutines can extract a week number from a date-time value. To determine a week number, they can use ISO 8601 standard week numbering, which 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.
These functions can also define the first week of the year as the first week in January with seven days. This is the definition they used in prior releases.

You specify which type of week numbering to use by setting the WEEKFIRST parameter.

Since the week number returned by HNAME 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 from a given date-time value.

**Syntax:** How to Set a Day as the Start of the Week

SET WEEKFIRST = {value|7}

where:

value

Can be:

1 through 7, representing Sunday through Saturday with non-standard week numbering.

or

ISO1 through ISO7, representing Sunday through Saturday with ISO standard week numbering.

**Note:** ISO is a synonym for ISO2.

The ISO standard establishes Monday as the first day of the week, so to be fully ISO compliant, the WEEKFIRST parameter should be set to ISO or ISO2.

**Example:** Setting Sunday as the Start of the Week

The following designates Sunday as the start of the week:

SET WEEKFIRST = 1

**Syntax:** How to View the Current Setting of WEEKFIRST

? SET WEEKFIRST

This returns the integer value of the first day of the week. For example, the integer 1 represents Sunday.
Controlling Processing of Date-Time Values

**How to:**
Enable Strict 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**

```plaintext
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

**Reference:**
Arguments for Use With Date and 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.
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>0.999999</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).
Using Date-Time Formats

In this section:

- Numeric String Format
- Formatted-string Format
- Translated-string Format
- Time Format

There are three types of date formats that are valid in date-time values: numeric string format, formatted-string format, and translated-string format. In each format, two-digit years are interpreted using the DEFCENT and YRTHRESH parameters.

Time components are separated by colons and may be followed by A.M., P.M., a.m., or p.m.

The DATEFORMAT parameter specifies the order of the date components (month/day/year) when date-time values are entered in the formatted string and translated string formats. It makes a value’s input format independent of the format of the variable to which it is being assigned.

Numeric String Format

The numeric string format is exactly two, four, six, or eight digits. Four-digit strings are considered to be a year (century must be specified), and the month and day are set to January 1. Six and eight-digit strings contain two or four digits for the year, followed by two for the month, and two for the day. Because the component order is fixed with this format, the DATEFORMAT setting is ignored.

If a numeric-string format longer than eight digits is encountered, it is treated as a combined date-time string in the Hnn format.

Example: Using Numeric String Format

The following are examples of numeric string date constants:

<table>
<thead>
<tr>
<th>String</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>January 1, 1999</td>
</tr>
<tr>
<td>1999</td>
<td>January 1, 1999</td>
</tr>
<tr>
<td>19990201</td>
<td>February 1, 1999</td>
</tr>
</tbody>
</table>
**Formatted-string Format**

The formatted-string format contains a one or two-digit day, a one or two-digit month, and a two or four-digit year, each component separated by a space, slash, hyphen, or period. All three components must be present and follow the DATEFORMAT setting. If any of the three fields is four digits, it is interpreted as the year, and the other two fields must follow the order given by the DATEFORMAT setting.

**Example: Using Formatted-string Format**

The following are examples of formatted-string date constants and specify May 20, 1999:

- 1999/05/20
- 5 20 1999
- 99.05.20
- 1999–05–20

**Translated-string Format**

The translated-string format contains the full or abbreviated month name. The year must also be present in four-digit or two-digit form. If the day is missing, day 1 of the month is assumed; if present, it can have one or two digits. If the string contains both a two-digit year and a two-digit day, they must be in the order given by the DATEFORMAT setting.

**Example: Using Translated-string Format**

The following date is in translated-string format:

- January 6 2000

**Time Format**

Time components are separated by colons and may be followed by A.M., P.M., a.m., or p.m. Seconds can be expressed with a decimal point or be followed by a colon. If there is a colon after seconds, the value following it represents milliseconds. There is no way to express microseconds or nanoseconds using this notation.

A decimal point in the seconds value indicates the decimal fraction of a second. Microseconds can be represented using six decimal digits. Nanoseconds can be represented using nine decimal digits.
**Example: Using Time Formats**

The following are examples of acceptable time formats:

- 14:30:20:99 (99 milliseconds)
- 14:30:20.99 (99/100 seconds)
- 14:30:20.999999 (999999 microseconds)
- 02:30:20:500pm

**Example: Using Universal Date-Time Input Values**

With DTSTANDARD settings of STANDARD and STANDARDU, the following date-time values can be read as input:

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:30[:20,99]</td>
<td>Comma separates time components instead of period</td>
</tr>
<tr>
<td>14:30[:20.99]Z</td>
<td>Universal time</td>
</tr>
<tr>
<td>15:30[:20,99]+01</td>
<td>Each of these is the same as above in Central European Time</td>
</tr>
<tr>
<td>15:30[:20,99]+0100</td>
<td></td>
</tr>
<tr>
<td>15:30[:20,99]+01:00</td>
<td></td>
</tr>
<tr>
<td>09:30[:20.99]-05</td>
<td>Same as above in Eastern Standard Time</td>
</tr>
</tbody>
</table>

Note that these values are stored identically internally with the STANDARDU setting. With the STANDARD setting, everything following the Z, +, or - is ignored.

**Assigning Date-Time Values**

**How to:**

Assign Date-Time Values

A date-time value is a constant in character format assigned by one of the following:

- A sequential data source.
- An expression that defines WHERE or IF criteria or creates a temporary field using the DEFINE or COMPUTE command.

A date-time constant can have blanks at the beginning or end or immediately preceding an am/pm indicator.
Syntax: How to Assign Date-Time Values

In a character file

\[date_string \ [time_string]\]

or

\[time_string \ [date_string]\]

In a COMPUTE, DEFINE, or WHERE expression

\[DT(date_string \ [time_string])\]

or

\[DT(time_string \ [date_string])\]

In an IF expression

\[date_string \ [time_string]\]

or

\[time_string \ [date_string]\]

where:

- **time_string**
  Is a time string in acceptable format. A time string can have a blank immediately preceding an am/pm indicator.

- **date_string**
  Is a date string in either numeric string, formatted-string, or translated-string format.

  In an IF criteria, if the value does not contain blanks or special characters, the single quotation marks are not necessary.

  **Note:** The date and time strings must be separated by at least one blank space. Blank spaces are also permitted at the beginning and end of the date-time string.

Example: Assigning Date-Time Literals

The DT prefix can be used in a COMPUTE, DEFINE, or WHERE expression to assign a date-time literal to a date-time field. For example:

\[DT2/HYYMDS = DT(20051226 05:45);\]
\[DT3/HYYMDS = DT(2005 DEC 26 05:45);\]
\[DT4/HYYMDS = DT(December 26 2005 05:45);\]
**Example:** Assigning a Date-Time Value in a COMPUTE Command

The following uses the DT function in a COMPUTE command to create a new field containing an assigned date-time value.

```plaintext
TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME AND COMPUTE
NEWSAL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
WHERE CURR_JOBCODE LIKE 'B%' 
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>

**Example:** Assigning a Date-Time Value in WHERE Criteria

The following uses the DT function to create a new field containing an assigned date-time value. This value is then used as a WHERE criteria.

```plaintext
DEFINE FILE EMPLOYEE
NEWSAL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME NEWSAL RAISETIME
WHERE RAISETIME EQ DT(20000101 09:00AM)
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$12,100.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$10,450.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$32,670.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$29,548.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$9,900.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$29,768.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>
**Example:** Assigning a Date-Time Value in IF Criteria

The following uses the DT function to create a new field containing an assigned date-time value. This value is then used as an IF criteria.

```plaintext
DEFINE FILE EMPLOYEE
NEWSAL/D12.2M = CURR_SAL + (0.1 * CURR_SAL);
RAISETIME/HYYMDIA = DT(20000101 09:00AM);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME NEWSAL RAISETIME
IF RAISETIME EQ '20000101 09:00AM'
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>NEWSAL</th>
<th>RAISETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$12,100.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$14,520.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$10,450.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$32,670.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$29,548.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$23,232.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>$20,328.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$23,958.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$17,710.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$9,900.00</td>
<td>2000/01/01 9:00AM</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$29,768.20</td>
<td>2000/01/01 9:00AM</td>
</tr>
</tbody>
</table>

**HADD: Incrementing a Date-Time Value**

**How to:**

Increment a Date-Time Value

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

**Syntax:**

How to Increment a Date-Time Value

```
HADD(datetime, 'component', increment, length, output)
```

where:

- **datetime**
  - Date-time

  Is the date-time value to be incremented, the name of a date-time field that contains the value, or an expression that returns the value.
**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 268.

**Note:** WEEKDAY is not a valid component for HADD.

**increment**

Integer

Is the number of units (positive or negative) by which to increment the component, the name of a numeric field that contains the value, or an expression that returns the value.

**length**

Integer

Is the number of characters returned. Valid values are:

- **8** indicates a date-time value that includes one to three decimal digits (milliseconds).
- **10** indicates a date-time value that includes four to six decimal digits (microseconds).
- **12** indicates a date-time value that includes seven to nine decimal digits (nanoseconds).

**output**

Date-time

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

**Example:**  **Incrementing the Month Component of a Date-Time Field (Reporting)**

HADD adds two months to each value in TRANSDATE and stores the result in ADD_MONTH. If necessary, the day is adjusted so that it is valid for the resulting month.

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ADD_MONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/04/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/08/26 05:45:00</td>
</tr>
</tbody>
</table>
Example: Incrementing the Month Component of a Date-Time Field (Maintain)

HADD adds two months to the DT1 field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID DT1 INTO DTSTK
COMPUTE
NEW_DATE/HYYMDS = HADD(DTSTK.DT1, 'MONTH', 2,10, NEW_DATE);
TYPE "DT1 IS: <DTSTK(1).DT1 "
TYPE "NEW_DATE IS: <NEW_DATE "
```

The result is:

DT1 IS: 2000/1/1 02:57:25
NEW_DATE IS: 2000/3/1 02:57:25
TRANSACTIONS: COMMITS = 1 ROLLBACKS = 0
SEGMENTS: INCLUDED = 0 UPDATED = 0 DELETED = 0

HCNVRT: Converting a Date-Time Value to Alphanumeric Format

How to:

Convert a Date-Time Value to Alphanumeric Format

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

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

```
HCNVRT(datetime, '(format)', length, output)
```

where:

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

- `format`
  - Alphanumeric
  - Is the format of the date-time field enclosed in parentheses and single quotation marks. It must be a date-time format (data type H, up to H23).
length

Integer

Is the number of characters in the alphanumeric field that is returned. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value. If length is smaller than the number of characters needed to display the alphanumeric field, the function returns a blank.

output

Alphanumeric

Is the field that contains the result, or the format of the output value enclosed in single quotation marks. This field must be in alphanumeric format and must be long enough to contain all of the characters returned.

Example: Converting a Date-Time Field to Alphanumeric Format (Reporting)

HCNVRT converts the TRANSDATE field to alphanumeric format. The first function does not include date-time display options for the field; the second function does for readability. It also specifies the display of seconds in the input field.

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ALPHA_DATE_TIME1/A20 = HCNVRT(TRANSDATE, '(H17)', 17, 'A20');
ALPHA_DATE_TIME2/A20 = HCNVRT(TRANSDATE, '(HYYMDS)', 20, 'A20');
WHERE DATE EQ 2000
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ALPHA_DATE_TIME1</th>
<th>ALPHA_DATE_TIME2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>200002050330000000</td>
<td>2000/02/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>200006260545000000</td>
<td>2000/06/26 05:45:00</td>
</tr>
</tbody>
</table>

Example: Converting a Date-Time Field to Alphanumeric Format (Maintain)

HCNVRT converts the DT1 field to alphanumeric format:

```
MAINTAIN FILE DATETIME
FOR ALL NEXT ID INTO STK;
COMPUTE
RESULT_HCNVRT/A20 = HCNVRT(STK.DT1, '(HYYMDH)', 20, RESULT_HCNVRT);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "RESULT_HCNVRT = " RESULT_HCNVRT;
END
```
**HDATE: Converting the Date Portion of a Date-Time Value to a Date Format**

**How to:**
Convert 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:**
How to Convert the Date Portion of a Date-Time Value to a Date Format

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

*where:*

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

- `output`  
  Date  
  Is the format in single quotation marks or the field that contains the result.

**Example:**
Converting the Date Portion of a Date-Time Field to a Date Format (Reporting)

HDATE converts the date portion of the TRANSDATE field to the date format YYMD:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_DATE/YYMD = HDATE(TRANSDATE, 'YYMD');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/02/05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/06/26</td>
</tr>
</tbody>
</table>
Example: Converting the Date Portion of a Date-Time Field to a Date Format (Maintain)

HDATE converts the date portion of DT1 to date format YYMD:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT1_DATE/YYMD = HDATE(STK.DT1, DT1_DATE);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE = <DT1_DATE";
END
```

The output is:

```
STK(1).DT1 = 2000/1/1 02:57:25
DT1_DATE = 2000/01/01
```

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

**How to:**

Find the Number of Units Between Two Date-Time Values

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

**Syntax:**

```
HDIFF(end_dt, start_dt, 'component', output)
```

where:

- **end_dt**
  - Date-time
  - Is the date-time value to subtract from, the name of a date-time field that contains the value, or an expression that returns the value.

- **start_dt**
  - Date-time
  - Is the date-time value to subtract, the name of a date-time field that contains the value, or an expression that returns the value.
**HDIFF: Finding the Number of Units Between Two Date-Time Values**

**component**

Alphanumeric

Is the name of the component to be used in the calculation, enclosed in single quotation marks. If the component is a week, the WEEKFIRST parameter setting is used in the calculation.

**output**

Floating-point double-precision

Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be floating-point double-precision.

**Example:** Finding the Number of Days Between Two Date-Time Fields (Reporting)

HDIFF calculates the number of days between the TRANSDATE and ADD_MONTH fields and stores the result in DIFF_PAYS, which has the format D12.2:

```plaintext
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
DIFF_DAYS/D12.2 = HDIFF(ADD_MONTH, TRANSDATE, 'DAY', 'D12.2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ADD_MONTH</th>
<th>DIFF_DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/04/05 03:30:00</td>
<td>60.00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/08/26 05:45:00</td>
<td>61.00</td>
</tr>
</tbody>
</table>

**Example:** Finding the Number of Days Between Two Date-Time Fields (Maintain)

HDIFF calculates the number of days between ADD_MONTH and DT1:

```plaintext
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
NEW_DATE/HYYMDS = HADD(STK.DT1, 'MONTH', 2,10, NEW_DATE);
DIFF_DAYS/D12.2 = HDIFF(NEW_DATE,STK.DT1,'DAY', DIFF_DAYS);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "NEW_DATE = "NEW_DATE;
TYPE "DIFF_DAYS = "DIFF_DAYS
END
```
HDTTM: Converting a Date Value to a Date-Time Value

**How to:**
Convert 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**

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

where:

**date**

Date

Is the date to be converted, the name of a date field that contains the value, or an expression that returns the value. It must be a full component format date. For example, it can be MDYY or YYJUL.

**length**

Integer

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

- 8 indicates a time value that includes milliseconds.
- 10 indicates a time value that includes microseconds.
- 12 indicates a time value that includes nanoseconds.

**output**

Date-time

Is the generated date-time value. It can be a field or the format of the output value enclosed in single quotation marks. The value must have a date-time format (data type H).
**Example:** Converting a Date Field to a Date-Time Field (Reporting)

HDTTM converts the date field TRANSDATE_DATE to a date-time field:

```plaintext
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_DATE/YYMD = HDATE(TRANSDATE, 'YYMD');
DT2/HYYMDIA = HDTTM(TRANSDATE_DATE, 8, 'HYYMDIA');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_DATE</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/02/05</td>
<td>2000/02/05 12:00AM</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/06/26</td>
<td>2000/06/26 12:00AM</td>
</tr>
</tbody>
</table>

**Example:** Converting a Date Field to a Date-Time Field (Maintain)

HDTTM converts the date field DT1_DATE to a date-time field:

```plaintext
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT1_DATE/YYMD = HDATE(DT1, DT1_DATE);
DT2/HYYMDIA = HDTTM(DT1_DATE, 8, DT2);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE = <DT1_DATE";
TYPE "DT2 = <DT2";
END
```

HEXTR: Extracting Components of a Date-Time Value and Setting Remaining Components to Zero

**How to:** Extract Multiple Components From a Date-Time Value

The HEXTR function extracts one or more components from a date-time value and moves them to a target date-time field with all other components set to zero.
**Syntax:**

How to Extract Multiple Components From a Date-Time Value

\[ \text{HEXTR}(\text{datetime}, \ 'componentstring', \ length, \ output) \]

where:

- **datetime**
  - Date-time
  - Is the a date-time value from which to extract the specified components.

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>century (the two high-order digits only of the four-digit year)</td>
</tr>
<tr>
<td>Y</td>
<td>year (the two low-order digits only of the four-digit year)</td>
</tr>
<tr>
<td>YY</td>
<td>Four digit year.</td>
</tr>
<tr>
<td>M</td>
<td>month</td>
</tr>
<tr>
<td>D</td>
<td>day</td>
</tr>
<tr>
<td>H</td>
<td>hour</td>
</tr>
<tr>
<td>I</td>
<td>minutes</td>
</tr>
<tr>
<td>S</td>
<td>seconds</td>
</tr>
<tr>
<td>s</td>
<td>milliseconds (the three high-order digits of the six-digit microseconds value)</td>
</tr>
<tr>
<td>u</td>
<td>microseconds (the three low-order digits of the six-digit microseconds value)</td>
</tr>
<tr>
<td>m</td>
<td>All six digits of the microseconds value.</td>
</tr>
<tr>
<td>n</td>
<td>Low order three digits of nine decimal digits.</td>
</tr>
</tbody>
</table>
**length**

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

- **8** - indicates a time value that includes milliseconds.
- **10** - indicates a time value that includes microseconds
- **12** - indicates a time value that includes nanoseconds

**output**

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

**Example: Extracting Hour and Minute Components Using HEXTR**

The VIDEOTR2 data source has a date-time field named TRANSDATE of type HYYMDI. The following request selects all records containing the time 09:18AM, regardless of the value of the remaining components:

```
TABLE FILE VIDEOTR2
PRINT TRANSDATE
BY LASTNAME
BY FIRSTNAME
WHERE HEXTR(TRANSDATE, 'HI', 8, 'HYYMDI') EQ DT(09:18AM)
END
```

The output is:

```
LASTNAME     FIRSTNAME   TRANSDATE
--------      ---------   ---------
DIZON         JANET       1999/11/05 09:18
PETERSON      GLEN        1999/09/09 09:18
```

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

**How to:**

Store the Current Date and Time in a Date-Time Field

The HGETC function returns the current 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 Date and Time in a Date-Time Field

HGETC(length, output)

where:

*length*

Integer

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

- 8 indicates a time value that includes milliseconds.
- 10 indicates a time value that includes microseconds.
- 12 indicates a time value that includes nanoseconds.

*output*

Date-time

Is the returned date-time value. Can be a field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

**Example:** Storing the Current Date and Time in a Date-Time Field (Reporting)

HGETC stores the current date and time in DT2:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DT2/HYYMDm = HGETC(10, 'HYYMDm');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/10/03 15:34:24.000000</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/10/03 15:34:24.000000</td>
</tr>
</tbody>
</table>

**Example:** Storing the Current Date and Time in a Date-Time Field (Maintain)

HGETC stores the current date and time in DT2:

```
MAINTAIN
COMPUTE DT2/HYYMDm = HGETC(10, DT2);
TYPE "DT2 = <DT2";
END
```
HHMMSS: Retrieving the Current Time

How to:
Retrieve the Current Time

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

A compiled MODIFY procedure must use HHMMSS to obtain the time; it cannot use the &TOD variable, which also returns the time. The &TOD variable is made current only when you execute a MODIFY, SCAN, or FSCAN procedure.

There is also an HHMMSS function available in the Maintain language. For information on this function, see HHMMSS: Retrieving the Current Time (Maintain) on page 306.

Syntax:   How to Retrieve the Current Time

HHMMSS(output)

where:

output
   Alphanumeric, at least A8
   Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Retrieving the Current Time

HHMMSS retrieves the current time and displays it in the page footing:

TABLE FILE EMPLOYEE
SUM CURR_SAL AS 'TOTAL SALARIES' AND COMPUTE
NOWTIME/A8 = HHMMSS(NOWTIME); NOPRINT
BY DEPARTMENT
FOOTING
"SALARY REPORT RUN AT TIME <NOWTIME"
END

The output is:

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>TOTAL SALARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$108,002.00</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>$114,282.00</td>
</tr>
</tbody>
</table>

SALARY REPORT RUN AT TIME 15.21.14
HINPUT: Converting an Alphanumeric String to a Date-Time Value

**How to:**
Convert 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

HINPUT(source_length, 'source_string', output_length, output)

where:

- **source_length**
  - Integer
  - Is the number of characters in the source string to be converted. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value.

- **source_string**
  - Alphanumeric
  - Is the string to be converted enclosed in single quotation marks, the name of an alphanumeric field that contains the string, or an expression that returns the string. The string can consist of any valid date-time input value.

- **output_length**
  - Integer
  - Is the length of the returned date-time value. Valid values are:
    - 8 indicates a time value that includes one to three decimal digits (milliseconds).
    - 10 indicates a time value that includes four to six decimal digits (microseconds).
    - 12 indicates a time value that includes seven to nine decimal digits (nanoseconds).

- **output**
  - Date-time
  - Is the returned date-time value. Is a field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).
**Example:** Converting an Alphanumeric String to a Date-Time Value (Reporting)

HCNVRT converts the TRANSDATE field to alphanumeric format, then HINPUT converts the alphanumeric string to a date-time value:

```plaintext
TABLE FILE VIDEOS tip AS 'DATE-TIME' AND COMPUTE
ALPHA_DATE_TIME/A20 = HCNVRT(TRANSDATE, '(H17)', 17, 'A20');
DT_FROM_ALPHA/HYYMDS = HINPUT(14, ALPHA_DATE_TIME, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>ALPHA_DATE_TIME</th>
<th>DT_FROM_ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>200002050300000000</td>
<td>2000/02/05 03:30:00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>200006260545000000</td>
<td>2000/06/26 05:45:00</td>
</tr>
</tbody>
</table>

**Example:** Converting an Alphanumeric String to a Date-Time Value (Maintain)

HINPUT converts the DT1 field to alphanumeric format:

```plaintext
MAINTAIN FILE DATETIME
COMPUTE
RESULT/HMtDYYmA = HINPUT(20,'19971029133059888999',10,RESULT);
TYPE RESULT;
END
```

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

**How to:**

Set 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

`HMIDNT(datetime, length, output)`

where:

- **datetime**
  
  Date-time
  
  Is the date-time value whose time is to be set to midnight, the name of a date-time field that contains the value, or an expression that returns the value.
**length**

Integer

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

- **8** indicates a time value that includes milliseconds.
- **10** indicates a time value that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

**output**

Date-time

Is the date-time return value whose time is set to midnight and whose date is copied from timestamp. Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

**Example: Setting the Time to Midnight (Reporting)**

HMINDT sets the time portion of the TRANSDATE field to midnight first in the 24-hour system and then in the 12-hour system:

```sql
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
TRANSDATE_MID_24/HYYMDS = HMINDT(TRANSDATE, 8, 'HYYMDS');
TRANSDATE_MID_12/HYYMDSA = HMINDT(TRANSDATE, 8, 'HYYMDSA');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>TRANSDATE_MID_24</th>
<th>TRANSDATE_MID_12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>2000/06/26 00:00:00</td>
<td>2000/06/26 12:00:00AM</td>
</tr>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>2000/02/05 00:00:00</td>
<td>2000/02/05 12:00:00AM</td>
</tr>
</tbody>
</table>

**Example: Setting the Time to Midnight (Maintain)**

HMINDT sets the time portion of DT1 to midnight in both the 24- and 12-hour systems:

```sql
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT_MID_24/HYYMDS = HMINDT(STK(1).DT1, 8, DT_MID_24);
DT_MID_12/HYYMDSA = HMINDT(STK(1).DT1, 8, DT_MID_12);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "DT_MID_24 = "DT_MID_24";
TYPE "DT_MID_12 = "DT_MID_12";
END
```
The HMASK function extracts one or more components from a date-time value and moves them to a target date-time field with all other components of the target field preserved.

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

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

where:

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>century (the two high-order digits only of the four-digit year)</td>
</tr>
<tr>
<td>Y</td>
<td>year (the two low-order digits only of the four-digit year)</td>
</tr>
<tr>
<td>YY</td>
<td>Four digit year.</td>
</tr>
<tr>
<td>M</td>
<td>month</td>
</tr>
<tr>
<td>D</td>
<td>day</td>
</tr>
<tr>
<td>H</td>
<td>hour</td>
</tr>
<tr>
<td>I</td>
<td>minutes</td>
</tr>
<tr>
<td>S</td>
<td>seconds</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>s</td>
<td>milliseconds (the three high-order digits of the six-digit microseconds value)</td>
</tr>
<tr>
<td>u</td>
<td>microseconds (the three low-order digits of the six-digit microseconds value)</td>
</tr>
<tr>
<td>m</td>
<td>All six digits of the microseconds value.</td>
</tr>
<tr>
<td>n</td>
<td>Low order three digits of nine decimal digits.</td>
</tr>
</tbody>
</table>

**input**

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

**length**

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

- 8 indicates a time value that includes 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**

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

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

HMASK processing is subject to the DTSTRICT setting. Moving the day (D) component without the month (M) component could lead to an invalid result, which is not permitted if the DTSTRICT setting is ON. Invalid date-time values cause any date-time function to return zeros.
**Example:**  **Changing a Date-Time Field Using HMASK**

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

```fortran
DEFINE FILE VIDEOTR2
ORIG_TRANSDATE/HYYMDI = TRANSDATE;
TRANSDATE =
  IF HEXTR(TRANSDATE, 'HI', 8, 'HHI') GT DT(12:00)
    THEN HADD (HMASK(DT(08:30), 'HISs', TRANSDATE, 8, 'HYYMDI'), 'DAY',
                  1,8, 'HYYMDI')
    ELSE TRANSDATE;
END

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

The output is

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>ORIG_TRANSDATE</th>
<th>TRANSDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERTAL</td>
<td>MARCIA</td>
<td>1999/07/29 12:19</td>
<td>1999/07/30 08:30</td>
</tr>
<tr>
<td>GARCIA</td>
<td>JOANN</td>
<td>1998/05/08 12:48</td>
<td>1998/05/09 08:30</td>
</tr>
<tr>
<td>PARKER</td>
<td>GLENNDA</td>
<td>1999/01/06 12:22</td>
<td>1999/01/07 08:30</td>
</tr>
<tr>
<td>RATHER</td>
<td>MICHAEL</td>
<td>1998/02/28 12:33</td>
<td>1998/03/01 08:30</td>
</tr>
<tr>
<td>WILSON</td>
<td>KELLY</td>
<td>1999/06/26 12:34</td>
<td>1999/06/27 08:30</td>
</tr>
</tbody>
</table>

---

**HNAME: Retrieving a Date-Time Component in Alphanumeric Format**

**How to:**

Retrieves a Date-Time Component in Alphanumeric Format

The HNAME function extracts a specified component from a date-time value and returns it as digits in alphanumeric format.
### Syntax: How to Retrieve a Date-Time Component in Alphanumeric Format

HNAME(\textit{datetime}, 'component', output)

where:

- **datetime**
  - Date-time
  - Is the date-time value from which a component value is to be extracted, the name of a date-time field containing the value that contains the value, or an expression that returns the value.

- **component**
  - Alphanumeric
  - Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid component names, see *Arguments for Use With Date and Time Functions* on page 268.

- **output**
  - Alphanumeric, at least A2
  - Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in alphanumeric format.
  - The function converts all other components to strings of digits only. The year is always four digits, and the hour assumes the 24-hour system.

### Example: Retrieving the Week Component in Alphanumeric Format (Reporting)

HNAME returns the week in alphanumeric format from the TRANSDATE field. Changing the WEEKFIRST parameter setting changes the value of the component.

```plaintext
SET WEEKFIRST = 7
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
    WEEK_COMPONENT/A10 = HNAME(TRANSDATE, 'WEEK', 'A10');
WHERE DATE EQ 2000;
END
```

When WEEKFIRST is set to seven, the output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>WEEK_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>06</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
</tbody>
</table>
When WEEKFIRST is set to three, the output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>WEEK_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>25</td>
</tr>
</tbody>
</table>

For details on WEEKFIRST, see the *Developing Applications* manual.

**Example: Retrieving the Day Component in Alphanumeric Format (Reporting)**

HNAME retrieves the day in alphanumeric format from the TRANSDATE field:

```plaintext
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DAY_COMPONENT/A2 = HNAME (TRANSDATE, 'DAY', 'A2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DAY_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>05</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
</tbody>
</table>

**Example: Retrieving the Day Component in Alphanumeric Format (Maintain)**

HNAME extracts the day in alphanumeric format from DT1:

```plaintext
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DAY_COMPONENT/A2 = HNAME (STK.DT1, 'DAY', DAY_COMPONENT);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "DAY_COMPONENT = <DAY_COMPONENT"
END
```

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

**How to:**

Retrieve a Date-Time Component in Numeric Format

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:

- **datetime**
  - Date-time
  - Is the date-time value from which the component is to be extracted, the name of a date-time field that contains the value, or an expression that returns the value.

- **component**
  - Alphanumeric
  - Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see Arguments for Use With Date and Time Functions on page 268.

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

**Example:** Retrieving the Day Component in Numeric Format (Reporting)

HPART retrieves the day in integer format from the TRANSDATE field:

```sql
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
DAY_COMPONENT/I2 = \text{HPART}(\text{TRANSDATE}, \ '\text{DAY}', \ 'I2');
WHERE DATE EQ 2000;
END
```

The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>DAY_COMPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>26</td>
</tr>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>5</td>
</tr>
</tbody>
</table>
Example:  Retriving the Day Component in Numeric Format (Maintain)

HPART extracts the day in integer format from DT1:

```plaintext
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
  DAY_COMPONENT/I2 = HPART(STK.DT1, 'DAY', DAY_COMPONENT);
  TYPE "STK(1).DT1 = <STK(1).DT1"
  TYPE "DAY_COMPONENT = <DAY_COMPONENT"
END
```

HSETPT: Inserting a Component Into a Date-Time Value

How to:

Insert a Component Into a Date-Time Value

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

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

HSETPT(datetime, 'component', value, length, output)

where:

- **datetime**
  Date-time
  Is the date-time value in which to insert the component, the name of a date-time field that contains the value, or an expression that returns the value.

- **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 268 for a list of valid components.

- **value**
  Integer
  Is the numeric value to be inserted for the requested component, the name of a numeric field that contains the value, or an expression that returns the value.
**length**

Integer

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

- **8** indicates a time value that includes one to three decimal digits (milliseconds).
- **10** indicates a time value that includes four to six decimal digits (microseconds).
- **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

**output**

Date-time

Is the returned date-time value whose chosen component is updated. All other components are copied from the source date-time value.

Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be in date-time format (data type H).

**Example:**  **Inserting the Day Component Into a Date-Time Field (Reporting)**

HSETPT inserts the day as 28 into the ADD_MONTH field and stores the result in INSERT_DAY:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
ADD_MONTH/HYYMDS = HADD(TRANSDATE, 'MONTH', 2, 8, 'HYYMDS');
INSERT_DAY/HYYMDS = HSETPT(ADD_MONTH, 'DAY', 28, 8, 'HYYMDS');
WHERE DATE EQ 2000;
END
```

The output is:

```
CUSTID  DATE-TIME         ADD_MONTH            INSERT_DAY
------  ---------         ---------            ----------
1118    2000/06/26 05:45  2000/08/26 05:45:00  2000/08/28 05:45:00
1237    2000/02/05 03:30  2000/04/05 03:30:00  2000/04/28 03:30:00
```

**Example:**  **Inserting the Day Component Into a Date-Time Field (Maintain)**

HSETPT inserts the day into ADD_MONTH:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
ADD_MONTH/HYYMDS = HADD(STK.DT1,'MONTH', 2, 8, ADD_MONTH);
INSERT_DAY/HYYMDS = HSETPT(ADD_MONTH, 'DAY', 28, 8, INSERT_DAY);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "ADD_MONTH = <ADD_MONTH";
TYPE "INSERT_DAY = <INSERT_DAY";
END
```
**HTIME: Converting the Time Portion of a Date-Time Value to a Number**

**How to:**
Convert 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, the name of a date-time field that contains the value, or an expression that returns the value.

- **output**
  Floating-point double-precision
  Is the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be floating-point double-precision.

**Example:**

**Converting the Time Portion of a Date-Time Field to a Number (Reporting)**

HTIME converts the time portion of the TRANSDATE field to the number of milliseconds:

```
TABLE FILE VIDEOTR2
PRINT CUSTID TRANSDATE AS 'DATE-TIME' AND COMPUTE
MILLISEC/D12.2 = HTIME(8, TRANSDATE, 'D12.2');
WHERE DATE EQ 2000;
END
```
The output is:

<table>
<thead>
<tr>
<th>CUSTID</th>
<th>DATE-TIME</th>
<th>MILLISEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>2000/02/05 03:30</td>
<td>12,600,000.00</td>
</tr>
<tr>
<td>1118</td>
<td>2000/06/26 05:45</td>
<td>20,700,000.00</td>
</tr>
</tbody>
</table>

**Example:** Converting the Time Portion of a Date-Time Field to a Number (Maintain)

HTIME converts the time portion of the DT1 field to the number of milliseconds:

```plaintext
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE MILLISEC/D12.2 = HTIME(8, STK.DT1, MILLISEC);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "MILLISEC = <MILLISEC";
END
```

**HTMTOTS or TIMETOTS: Converting a Time to a Timestamp**

**How to:**

Convert a Time to a Timestamp

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

**Note:** TIMETOTS is a synonym for HTMTOTS.

**Syntax:**

**How to Convert a Time to a Timestamp**

HTMTOTS(time, length, output)

or

TIMETOTS(time, length, output)

where:

time

Date-Time

Is the date-time value whose time will be used. The date portion will be ignored.
**length**
- Integer
  - Is the length of the result. This can be one of the following:
    - 8 for input time values including milliseconds.
    - 10 for input time values including microseconds.
    - 12 for input time values including nanoseconds.

**output_format**
- Date-Time
  - Is the timestamp whose date is set to current date, and whose time is copied from time.
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

**Example: Converting a Time to a Timestamp**

HTMTOTS converts the time portion of the TRANSDATE field to a timestamp, using the current date for the date portion of the returned value:

```plaintext
DEFINE FILE VIDEOTR2
  TSTMPSEC/HYYMDS = HTMTOTS(TRANSDATE, 8, 'HYYMDS');
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE TSTMPSEC
BY LASTNAME BY FIRSTNAME
WHERE DATE EQ '1991'
END
```

The output is:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>TRANSDATE</th>
<th>TSTMPSEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRUZ</td>
<td>IVY</td>
<td>1991/06/27 02:45</td>
<td>2011/01/11 02:45:00</td>
</tr>
<tr>
<td>GOODMAN</td>
<td>JOHN</td>
<td>1991/06/25 01:19</td>
<td>2011/01/11 01:19:00</td>
</tr>
<tr>
<td>GREEVEN</td>
<td>GEORGIA</td>
<td>1991/06/24 10:27</td>
<td>2011/01/11 10:27:00</td>
</tr>
<tr>
<td>HANDLER</td>
<td>EVAN</td>
<td>1991/06/20 05:15</td>
<td>2011/01/11 05:15:00</td>
</tr>
<tr>
<td>KRAMER</td>
<td>CHERYL</td>
<td>1991/06/21 07:11</td>
<td>2011/01/11 07:11:00</td>
</tr>
<tr>
<td>MONROE</td>
<td>CATHERINE</td>
<td>1991/06/21 01:10</td>
<td>2011/01/11 01:10:00</td>
</tr>
<tr>
<td>SPIVEY</td>
<td>TOM</td>
<td>1991/06/21 07:18</td>
<td>2011/01/11 07:18:00</td>
</tr>
<tr>
<td>WILLIAMS</td>
<td>KENNETH</td>
<td>1991/06/21 04:11</td>
<td>2011/01/11 04:11:00</td>
</tr>
<tr>
<td></td>
<td>PATRICK</td>
<td>1991/06/24 01:17</td>
<td>2011/01/11 01:17:00</td>
</tr>
<tr>
<td></td>
<td>SPIVEY</td>
<td>1991/06/24 01:17</td>
<td>2011/01/11 01:17:00</td>
</tr>
<tr>
<td></td>
<td>WILLIAMS</td>
<td>1991/06/24 04:43</td>
<td>2011/01/11 04:43:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1991/06/24 02:08</td>
<td>2011/01/11 02:08:00</td>
</tr>
</tbody>
</table>
HYYWD: Returning the Year and Week Number From a Date-Time Value

**How to:***

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

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

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

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

**Syntax:**

How to Return the Year and Week Number From a Date-Time Value

HYYWD(dtvalue, output)

where:

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

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

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

yyyy-Www-d

where:

- **yyyy**
  - Is the four-digit year.

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

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

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

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

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

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

The output is:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>TRANSTOT</th>
<th>QUANTITY</th>
<th>ISODATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991/06/24 04:43</td>
<td>16.00</td>
<td></td>
<td>1991-W26-1</td>
</tr>
<tr>
<td>1991/06/27 02:45</td>
<td>16.00</td>
<td></td>
<td>1991-W26-4</td>
</tr>
<tr>
<td>1996/08/17 05:11</td>
<td>5.18</td>
<td></td>
<td>1996-W33-6</td>
</tr>
<tr>
<td>1998/02/04 04:11</td>
<td>12.00</td>
<td></td>
<td>1998-W06-3</td>
</tr>
<tr>
<td>1999/01/30 04:16</td>
<td>13.00</td>
<td></td>
<td>1999-W04-6</td>
</tr>
<tr>
<td>1999/04/22 06:19</td>
<td>3.75</td>
<td></td>
<td>1999-W16-4</td>
</tr>
<tr>
<td>1999/05/06 05:14</td>
<td>1.00</td>
<td></td>
<td>1999-W18-4</td>
</tr>
<tr>
<td>1999/08/09 03:17</td>
<td>15.00</td>
<td></td>
<td>1999-W32-1</td>
</tr>
<tr>
<td>1999/09/09 09:18</td>
<td>14.00</td>
<td></td>
<td>1999-W36-4</td>
</tr>
<tr>
<td>1999/10/16 09:11</td>
<td>5.18</td>
<td></td>
<td>1999-W41-6</td>
</tr>
<tr>
<td>1999/11/05 11:12</td>
<td>2.50</td>
<td></td>
<td>1999-W44-5</td>
</tr>
<tr>
<td>1999/12/09 09:47</td>
<td>5.18</td>
<td></td>
<td>1999-W49-4</td>
</tr>
<tr>
<td>1999/12/15 04:04</td>
<td>2.50</td>
<td></td>
<td>1999-W50-3</td>
</tr>
</tbody>
</table>
**Example: Extracting a Component From a Date Returned by HYYWD**

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

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

The output is:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>TRANSTOT</th>
<th>QUANTITY</th>
<th>ISODATE</th>
<th>WEEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991/06/24 04:43</td>
<td>16.00</td>
<td></td>
<td>1991-W26-1</td>
<td>26</td>
</tr>
<tr>
<td>1991/06/25 01:17</td>
<td>2.50</td>
<td></td>
<td>1991-W26-2</td>
<td>26</td>
</tr>
<tr>
<td>1991/06/27 02:45</td>
<td>16.00</td>
<td></td>
<td>1991-W26-4</td>
<td>26</td>
</tr>
</tbody>
</table>
Maintain-specific date and time functions manipulate date and time values. These functions are available only in Maintain.

There are additional date and time functions available in both the reporting and Maintain languages. For more information on these functions, see *Date-Time Functions* on page 263.
Maintain-specific Standard Date and Time Functions

In this section:

- HHMMSS: Retrieving the Current Time (Maintain)
- Initial_HHMMSS: Returning the Time the Application Was Started
- Initial_TODAY: Returning the Date the Application Was Started
- TODAY: Retrieving the Current Date (Maintain)
- TODAY2: Returning the Current Date
- ADD: Adding Days to a Date
- DAY: Extracting the Day of the Month From a Date
- JULIAN: Determining How Many Days Have Elapsed in the Year
- MONTH: Extracting the Month From a Date
- QUARTER: Determining the Quarter
- SETMDY: Setting the Value to a Date
- SUB: Subtracting a Value From a Date
- WEEKDAY: Determining the Day of the Week for a Date
- YEAR: Extracting the Year From a Date

Standard date and time functions are for use with non-legacy dates. For a definition of standard dates and times, see Date-Time Functions on page 263.

HHMMSS: Retrieving the Current Time (Maintain)

**How to:**

Retrieve the Current Time

The HHMMSS function retrieves the current time from the operating system as an 8-character string, separating the hours, minutes, and seconds with periods.

To use this function, you must import the function library MNTUWS. For information on importing a function library, see Calling a Function on page 44.

There is also an HHMMSS function available in the reporting language. For information on this function, see HHMMSS: Retrieving the Current Time on page 286.
**Syntax:** How to Retrieve the Current Time

HHMMSS()

**Example:** Retrieving the Current Time

HHMMSS retrieves the current time from the operating system:

MAINTAIN
Module Import (mntuws);
Case Top
Compute now/a10 = hhmmss();
type "Now = <<now"
EndCase
END

The output is:

Now = 14.25.33

**Initial_HHMMSS:** Returning the Time the Application Was Started

**How to:**

Retrieve the Initial Time

The Initial_HHMMSS function returns the time when the Maintain application was started as an 8-character string, with embedded periods separating the hours, minutes, and seconds.

To use this function, you must import the function library MNTUWS. For details on importing this library, see *Calling a Function* on page 44.

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

Initial_HHMMSS()

**Initial_TODAY:** Returning the Date the Application Was Started

**How to:**

Retrieve the Initial Date

The Initial_TODAY function returns the date in MM/DD/YY format when the Maintain application was started as an 8-character string with embedded slashes.

To use this function, you must import the function library MNTUWS. For details on importing this library, see *Calling a Function* on page 44.
How to Retrieve the Initial Date

Syntax:

Initial_TODAY()

TODAY: Retrieving the Current Date (Maintain)

How to:

Retrieve the Current Date

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

To use this function, you must import the function library MNTUWS. For information on importing this library, see Calling a Function on page 44.

There is a version of the TODAY function that is available only in the reporting language. For information on this function, see HTMTOTS or TIMETOTS: Converting a Time to a Timestamp on page 299.

Syntax: How to Retrieve the Current Date

TODAY()

Example: Retrieving the Current Date

TODAY retrieves the current date from the system:

MAINTAIN
Module Import (mntuws);

Case Top
Compute date1/a8 = today();
type "Date1 = <<date1"
Endcase
END

The result is:

Date1 = 07/17/02
TODAY2: Returning the Current Date

How to:
Retrieve the Current Date

The TODAY2 function retrieves the current date from the operating system in the format MM/DD/YYYY. Use format A10 with the TODAY2 function to ensure proper results.

To use this function, you must import the function library MNTUWS. For information on importing this library, see Calling a Function on page 44.

Syntax: How to Retrieve the Current Date

TODAY2()  

Example: Retrieving the Current Date

TODAY2 retrieves the current date from the system:

MAINTAIN
Module Import (mntuws);

Case Top
Compute date2/a10 = today2();
type "Date2 = <<date2"
Endcase
END

The result is:
Date2 = 07/17/2002

ADD: Adding Days to a Date

How to:
Add Days to a Date

The ADD function adds a given number of days to a date.
How to Add Days to a Date

Syntax:

ADD(date, value)

or

date.ADD(value)

where:

date
Is the date to add days to, or a field containing the date.

value
Is the number of days by which to increase the date.

This function changes the value of date.

Example: Adding Days to a Date

ADD adds 10 days to the each value in the DateVar field:

ADD(DateVar, 10)

The following are sample values for DateVar and the corresponding values for ADD(DateVar, 10):

<table>
<thead>
<tr>
<th>DateVar</th>
<th>ADD(DateVar, 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/1999</td>
<td>01/10/2000</td>
</tr>
<tr>
<td>01/01/2000</td>
<td>01/11/2000</td>
</tr>
<tr>
<td>01/02/2000</td>
<td>01/12/2000</td>
</tr>
</tbody>
</table>

DAY: Extracting the Day of the Month From a Date

How to:

Extract the Day of the Month From a Date

The DAY function extracts the day of the month from a date and returns the result as an integer.

Syntax: How to Extract the Day of the Month From a Date

DAY(date);

where:

date
Is the date (in date format) from which to extract the day of the month, or a field containing the date.
**Example:** **Extracting the Day of the Month From a Date**

DAY extracts the day of the month from the DATE field:

\[
\text{DAY(DATE)}
\]

The following are sample values for DATE and the corresponding values for DAY(DATE):

<table>
<thead>
<tr>
<th>DATE</th>
<th>DAY(DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>1</td>
</tr>
<tr>
<td>01/02/2000</td>
<td>2</td>
</tr>
<tr>
<td>01/03/2000</td>
<td>3</td>
</tr>
</tbody>
</table>

**JULIAN: Determining How Many Days Have Elapsed in the Year**

**How to:**

Determine How Many Days Have Elapsed in the Year

The JULIAN function determines the number of days that have elapsed in the given year up to a given date, and returns the result as an integer.

**Syntax:** **How to Determine How Many Days Have Elapsed in the Year**

\[
\text{JULIAN(date)};
\]

where:

\[
\text{date}
\]

Is the date (in date format) for which to determine the number of days elapsed in the given year, or a field containing the date.

**Example:** **Determining How Many Days Have Elapsed in the Year**

JULIAN determines the number of days that have elapsed up to the date in the DATE field:

\[
\text{JULIAN(DATE)}
\]

The following are sample values for DATE and the corresponding values for JULIAN(DATE):

<table>
<thead>
<tr>
<th>DATE</th>
<th>JULIAN(DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>1</td>
</tr>
<tr>
<td>02/01/2000</td>
<td>32</td>
</tr>
<tr>
<td>03/01/2000</td>
<td>61</td>
</tr>
</tbody>
</table>
MONTH: Extracting the Month From a Date

**How to:**
Extract the Month From a Date

The MONTH function extracts the month from a date and returns the result as an integer.

**Syntax:**

\[
\text{MONTH}(\text{date})
\]

where:

- **date**
  
  Is the date (in date format) from which to extract the month, or a field containing the date.

**Example:**

Extracting the Month From a Date

MONTH extracts the month from each value in the DATE field:

\[
\text{MONTH(DATE)}
\]

The following are sample values for DATE and the corresponding values for MONTH(DATE):

<table>
<thead>
<tr>
<th>DATE</th>
<th>MONTH (DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>1</td>
</tr>
<tr>
<td>02/01/2000</td>
<td>2</td>
</tr>
<tr>
<td>03/01/2000</td>
<td>3</td>
</tr>
</tbody>
</table>

QUARTER: Determining the Quarter

**How to:**

Determine the Quarter for a Date

The QUARTER function determines the quarter of the year in which a date resides, and returns the result as an integer.
Syntax: How to Determine the Quarter for a Date

Quarter(date);

Where:

date
Is the date (in date format) for which to determine the quarter, or a field containing the date.

Example: Determining the Quarter for a Date

Quarter extracts the quarter component from each value in the DATE field:

Quarter(DATE)

The following are sample values for DATE and the corresponding values for Quarters:

<table>
<thead>
<tr>
<th>DATE</th>
<th>Quarters</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>1</td>
</tr>
<tr>
<td>04/01/2000</td>
<td>2</td>
</tr>
<tr>
<td>07/01/2000</td>
<td>3</td>
</tr>
</tbody>
</table>

SETMDY: Setting the Value to a Date

How to:

Set a Value to a Date

The SETMDY function sets a value to a date based on numeric values representing a day, month, and year. SETMDY returns a 0 if the function is successful, and a negative number if the function fails.

Syntax: How to Set a Value to a Date

Setmdy(date, month, day, year);

Or

date.Setmdy(month, day, year);

Where:

date
Is the date, in date format, or a field containing the date.

Month
Is an integer value representing a month.
**Example: Setting a Value to a Date**

SETMDY sets the value of DateVar, which is formatted as a date that appears as wrMtrDYY (for example, Saturday, January 1, 2000):

```
SETMDY(DateVar, month, day, year);
```

The following are sample values for month, day, and year, and the corresponding dates for DateVar:

<table>
<thead>
<tr>
<th>month</th>
<th>day</th>
<th>year</th>
<th>DateVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>05</td>
<td>1965</td>
<td>Monday, April 5, 1965</td>
</tr>
<tr>
<td>02</td>
<td>01</td>
<td>1997</td>
<td>Saturday, February 1, 1997</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>2000</td>
<td>Saturday, January 1, 2000</td>
</tr>
</tbody>
</table>

**SUB: Subtracting a Value From a Date**

**How to:**

Subtract a Value From a Date

The SUB function subtracts a given number of days from a date.

**Syntax:**

How to Subtract a Value From a Date

```
SUB(date, value)
```

or

```
date.SUB(value)
```

where:

- **date**
  Is the date from which to subtract the value, or a field containing the date.

- **value**
  Is the value to subtract from the date.
Example: Subtracting Days From a Date

SUB subtracts 10 days from each value in the DateVar field.

SUB(DateVar, 10)

The following are sample values for DateVar and the corresponding values for SUB(DateVar, 10):

<table>
<thead>
<tr>
<th>DateVar</th>
<th>SUB(DateVar, 10);</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/1999</td>
<td>12/21/2000</td>
</tr>
<tr>
<td>01/01/2000</td>
<td>12/22/2000</td>
</tr>
<tr>
<td>01/02/2000</td>
<td>12/23/2000</td>
</tr>
</tbody>
</table>

WEEKDAY: Determining the Day of the Week for a Date

How to:

Determine the Day of the Week for a Date

The WEEKDAY function determines the day of the week for a date and returns the result as an integer (1=Monday, 2=Tuesday, and so on).

Syntax: How to Determine the Day of the Week for a Date

WEEKDAY(date);

where:

date

Is the date (in date format) for which to determine the weekday, or a field containing the date.

Example: Determining the Day of the Week for a Date

WEEKDAY determines the day of the week for each date in the DATE field, and stores that day as a number corresponding to a weekday:

WEEKDAY(DATE)

The following are sample values for DATE and the corresponding values for WEEKDAY(DATE):

<table>
<thead>
<tr>
<th>DATE</th>
<th>WEEKDAY (DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>6</td>
</tr>
<tr>
<td>01/02/2000</td>
<td>7</td>
</tr>
<tr>
<td>01/03/2000</td>
<td>1</td>
</tr>
</tbody>
</table>
YEAR: Extracting the Year From a Date

How to: Extract the Year From a Date

The YEAR function extracts the year from a date.

Syntax: How to Extract the Year From a Date

YEAR(date);

where:

  date
  Is the date from which to extract the year, or a field containing the date.

Example: Extracting a Year From a Date

YEAR extracts the year from the DATE field, and stores that year in the YEAR(DATE) field:

YEAR(DATE)

The following are sample values for DATE and the corresponding values for YEAR(DATE):

<table>
<thead>
<tr>
<th>DATE</th>
<th>YEAR(DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01/2000</td>
<td>2000</td>
</tr>
<tr>
<td>02/01/2001</td>
<td>2001</td>
</tr>
<tr>
<td>03/01/2002</td>
<td>2002</td>
</tr>
</tbody>
</table>
Format conversion functions convert fields from one format to another.

For information on field formats see the Describing Data manual.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function on page 43.

**Topics:**
- ATODBL: Converting an Alphanumeric String to Double-Precision Format
- EDIT: Converting the Format of a Field
- FPRINT: Converting Fields to Alphanumeric Format
- FTOA: Converting a Number to Alphanumeric Format
- HEXBYT: Converting a Decimal Integer to a Character
- ITONUM: Converting a Large Binary Integer 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
- 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**

**How to:**
Convert an Alphanumeric String to Double-Precision Format

The ATODBL function converts a number in alphanumeric format to decimal (double-precision) format.

**Syntax:**

ATODBL(source_string, length, output)

where:

source_string
Alphanumeric
Is the string consisting of digits and, optionally, one sign and one decimal point to be converted, or a field or variable that contains the string.

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

output
Double precision floating-point
Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

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

ATODBL converts the EMP_ID field into double-precision format and stores the result in D_EMP_ID:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME AND
EMP_ID AND
COMPUTE D_EMP_ID/D12.2 = ATODBL(EMP_ID, '09', D_EMP_ID);
WHERE DEPARTMENT EQ 'MIS';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>D_EMP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612</td>
<td>112,847,612.00</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>117,593,129.00</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>219,984,371.00</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>326,179,357.00</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>543,729,165.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>818,692,173.00</td>
</tr>
</tbody>
</table>

**Example:** Converting an Alphanumeric Value to Double-Precision Format With MODIFY

In the following example, the Master File contains the MISSING attribute for the CURR_SAL field. If you do not enter a value for this field, it is interpreted as the default value, a period.

FILENAME=EMPLOYEE, SUFFIX=FOC
SEGNAME=EMPINFO, SEGTYPE=S1
FIELDNAME=EMP_ID, ALIAS=EID, FORMAT=A9, $
  .
  .
  .
FIELDNAME=CURR_SAL, ALIAS=CSAL, FORMAT=D12.2M, MISSING=ON, $
  .
  .
  .

ATODBL converts the value supplied for TCSAL to double-precision format:

MODIFY FILE EMPLOYEE
  COMPUTE TCSAL/A12=;
  PROMPT EID
  MATCH EID
  ON NOMATCH REJECT
  ON MATCH TYPE "EMPLOYEE <D.LAST_NAME <D.FIRST_NAME"
  ON MATCH TYPE "ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE"
  ON MATCH PROMPT TCSAL
  ON MATCH COMPUTE
  CSAL MISSING ON = IF TCSAL EQ 'N/A' THEN MISSING
  ELSE ATODBL(TCSAL, '12', 'D12.2');
  ON MATCH TYPE "SALARY NOW <CSAL"
  DATA
A sample execution is:

```
EMPLOYEE    ON 11/14/96 AT 13.42.55
DATA FOR TRANSACTION    1
EMP_ID      = 071382660
EMPLOYEE STEVENS ALFRED
ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE
TCSAL       = N/A
SALARY NOW              .
DATA FOR TRANSACTION    2
EMP_ID      = 112847612
EMPLOYEE SMITH MARY
ENTER CURRENT SALARY OR 'N/A' IF NOT AVAILABLE
TCSAL       = 45000
SALARY NOW      $45,000.00
DATA FOR TRANSACTION    3
EMP_ID      = end
TRANSACTIONS:         TOTAL =     2  ACCEPTED=     2  REJECTED=     0
SEGMENTS:             INPUT =     0  UPDATED =     0  DELETED =     0
```

The procedure processes as follows:

1. For the first transaction, the procedure prompts for an employee ID. You enter 071382660.
2. The procedure displays the last and first name of the employee, STEVENS ALFRED.
3. The procedure prompts for a current salary. You enter N/A.
4. A period displays.
5. For the second transaction, the procedure prompts for an employee ID. You enter 112847612.
6. The procedure displays the last and first name of the employee, SMITH MARY.
7. Then it prompts for a current salary. Enter 45000.
8. $45,000.00 displays.
EDIT: Converting the Format of a Field

How to:
Convert 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 85.

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. CHGDAT is then able to use the field, which it expects in alphanumeric format:

```
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AND COMPUTE
ALPHA_HIRE/A17 = EDIT(HIRE_DATE); NOPRINT AND COMPUTE
HIRE_MDY/A17 = CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17');
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```

The output is:

```
LAST_NAME  FIRST_NAME  HIRE_DATE  HIRE_MDY
---------  ----------  ---------  --------
BLACKWOOD  ROSEMARIE   82/04/01   APRIL 01 1982
CROSS      BARBARA     81/11/02   NOVEMBER 02 1981
GREENSPAN  MARY        82/04/01   APRIL 01 1982
JONES      DIANE       82/05/01   MAY 01 1982
MCCOY      JOHN        81/07/01   JULY 01 1981
SMITH      MARY        81/07/01   JULY 01 1981
```

FPRINT: Converting Fields to Alphanumeric Format

How to: Convert Fields Using FPRINT

Reference: Usage Notes for the FPRINT Function

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.

output
Alphanumeric

Is the name of the output field or its format enclosed in single quotation marks.

The output format must be long enough to hold the converted number itself, with a sign and decimal point, plus any additional characters generated by display options, such as commas, a currency symbol, or a percent sign.

For example, D12.2 format is converted to A14 because it outputs two decimal digits, a decimal point, a possible minus sign, up to eight integer digits, and two commas. If the output format is not large enough, excess right-hand characters may be truncated.

Reference: Usage Notes for the FPRINT Function

- The output of FPRINT for numeric values is right-justified within the area required for the maximum number of characters corresponding to the supplied format. This ensures that all possible values are aligned vertically along the decimal point or units digit.

- By default, the column title is left justified for alphanumeric fields. To right justify the column title, use the /R reformatting option for the field.

- Maintain does not support the FPRINT function. However, you can do the same type of conversion in Maintain using the COMPUTE command.
Example: Converting Numeric Fields to Alphanumeric Format

The following request against the EMPLOYEE data source uses FPRINT to convert the CURR_SAL, ED_HRS, and BANK_ACCT fields to alphanumeric for display on the report output. Then, the STRREP function replaces the blanks in the alphanumeric representation of CURR_SAL with asterisks. CURR_SAL has format D12.2M, so the alphanumeric representation has format A15. The ED_HRS field has format F6.2, so the alphanumeric representation has format A6. The BANK_ACCT field has format I9S, so the alphanumeric representation has format A9. The alphanumeric representations of the numeric fields are right-justified. The /R options in the PRINT command cause the column titles to be right-justified over the values:

DEFINE FILE EMPLOYEE
ASAL/A15 = FPRINT(CURR_SAL, 'D12.2M', ASAL);
ASAL/A15 = STRREP(15, ASAL, 1, ' ', 1, '*', 15, ASAL);
AED/A6 = FPRINT(ED_HRS, 'F6.2', AED);
ABANK/A9 = FPRINT(BANK_ACCT, 'I9S', ABANK);
END
TABLE FILE EMPLOYEE
PRINT CURR_SAL ASAL
ED_HRS AED/R
BANK_ACCT ABANK/R
WHERE BANK_NAME NE ' '
ON TABLE SET PAGE NOPAGE
END

The output is:

<table>
<thead>
<tr>
<th>CURR_SAL</th>
<th>ASAL</th>
<th>ED_HRS</th>
<th>AED</th>
<th>BANK_ACCT</th>
<th>ABANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$18,480.00</td>
<td>*****$18,480.00</td>
<td>50.00</td>
<td>50.00</td>
<td>40950036</td>
<td>40950036</td>
</tr>
<tr>
<td>$29,700.00</td>
<td>*****$29,700.00</td>
<td>.00</td>
<td>.00</td>
<td>160633</td>
<td>160633</td>
</tr>
<tr>
<td>$26,862.00</td>
<td>*****$26,862.00</td>
<td>30.00</td>
<td>30.00</td>
<td>819000702</td>
<td>819000702</td>
</tr>
<tr>
<td>$21,780.00</td>
<td>*****$21,780.00</td>
<td>75.00</td>
<td>75.00</td>
<td>122850108</td>
<td>122850108</td>
</tr>
<tr>
<td>$16,100.00</td>
<td>*****$16,100.00</td>
<td>50.00</td>
<td>50.00</td>
<td>136500120</td>
<td>136500120</td>
</tr>
<tr>
<td>$27,062.00</td>
<td>*****$27,062.00</td>
<td>45.00</td>
<td>45.00</td>
<td>163800144</td>
<td>163800144</td>
</tr>
</tbody>
</table>
Example: Converting Alphanumeric and Numeric Date Fields to Alphanumeric Format

The following request against the EMPLOYEE data source converts the HIRE_DATE field to alphanumeric format. It also creates an alphanumeric date field named ADATE and converts it to its alphanumeric representation. The HIRE_DATE field has format I6YMD and the ADATE field has format A6YMD, so the alphanumeric representations have format A8 to account for the slashes between the date components. The /R option right-justifies the column titles over the field values:

```
DEFINE FILE EMPLOYEE
AHDATE/A8 = FPRINT(HIRE_DATE,'I6YMD', AHDATE);
ADATE/A6YMD = EDIT(HIRE_DATE);
AADATE/A8 = FPRINT(ADATE,'A6YMD', AADATE);
END
TABLE FILE EMPLOYEE
PRINT HIRE_DATE AHDATE/R
ADATE ADATE/R
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
HIRE_DATE    AHDATE  ADATE       ADATE
---------  --------  -----     --------
80/06/02  80/06/02  80/06/02  80/06/02
81/07/01  81/07/01  81/07/01  81/07/01
82/05/01  82/05/01  82/05/01  82/05/01
82/01/04  82/01/04  82/01/04  82/01/04
82/08/01  82/08/01  82/08/01  82/08/01
82/01/04  82/01/04  82/01/04  82/01/04
82/07/01  82/07/01  82/07/01  82/07/01
81/07/01  81/07/01  81/07/01  81/07/01
82/04/01  82/04/01  82/04/01  82/04/01
82/02/02  82/02/02  82/02/02  82/02/02
82/04/01  82/04/01  82/04/01  82/04/01
81/11/02  81/11/02  81/11/02  81/11/02
```

Using Functions 325
Example: Converting a Date Field to Alphanumeric Format

The following request against the VIDEOTRK data source converts the TRANSDATE (YMD) field to alphanumeric format. The alphanumeric representation has format A8 to account for the slashes between the date components:

```
DEFINE FILE VIDEOTRK
ALPHA_DATE/A8  = FPRINT(TRANSDATE,'YMD', ALPHA_DATE);
END
TABLE FILE VIDEOTRK
PRINT TRANSDATE ALPHA_DATE
WHERE TRANSDATE LE '91/06/20'
ON TABLE SET PAGE NOPAGE
END
```

The output is:

```
+---------+-----------+
| TRANSDATE | ALPHA_DATE |
|----------+-----------|
| 91/06/20 | 91/06/20  |
| 91/06/19 | 91/06/19  |
| 91/06/18 | 91/06/18  |
| 91/06/19 | 91/06/19  |
| 91/06/18 | 91/06/18  |
| 91/06/19 | 91/06/19  |
| 91/06/17 | 91/06/17  |
| 91/06/20 | 91/06/20  |
| 91/06/19 | 91/06/19  |
| 91/06/20 | 91/06/20  |
| 91/06/18 | 91/06/18  |
| 91/06/17 | 91/06/17  |
| 91/06/17 | 91/06/17  |
| 91/06/19 | 91/06/19  |
| 91/06/17 | 91/06/17  |
```
**Example:** Converting a Date-Time Field to Alphanumeric Format and Creating a HOLD File

The following request against the VIDEOTR2 data source converts the TRANSDATE (HYYMDI) field to alphanumeric format. The alphanumeric representation has format A16 to account for a four-digit year, two-digit month, two-digit day, two slashes between the date components, a space between the date and time, a two-digit hour, a colon between the hour and minute components, and a two-digit minute:

```plaintext
DEFINE FILE VIDEOTR2
DATE/I4 = HPART(TRANSDATE, 'YEAR', 'I4');
ALPHA_DATE/A16 = FPRINT(TRANSDATE,'HYYMDI', ALPHA_DATE);
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE ALPHA_DATE/R
WHERE DATE EQ '1991'
ON TABLE SET PAGE NOPAGE
END

The output is:

<table>
<thead>
<tr>
<th>TRANSDATE</th>
<th>ALPHA_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991/06/27 02:45</td>
<td>1991/06/27 02:45</td>
</tr>
<tr>
<td>1991/06/20 05:15</td>
<td>1991/06/20 05:15</td>
</tr>
<tr>
<td>1991/06/21 07:11</td>
<td>1991/06/21 07:11</td>
</tr>
<tr>
<td>1991/06/21 01:10</td>
<td>1991/06/21 01:10</td>
</tr>
<tr>
<td>1991/06/19 07:18</td>
<td>1991/06/19 07:18</td>
</tr>
<tr>
<td>1991/06/19 04:11</td>
<td>1991/06/19 04:11</td>
</tr>
<tr>
<td>1991/06/25 01:19</td>
<td>1991/06/25 01:19</td>
</tr>
<tr>
<td>1991/06/24 04:43</td>
<td>1991/06/24 04:43</td>
</tr>
<tr>
<td>1991/06/24 02:08</td>
<td>1991/06/24 02:08</td>
</tr>
<tr>
<td>1991/06/25 01:17</td>
<td>1991/06/25 01:17</td>
</tr>
<tr>
<td>1991/06/27 01:17</td>
<td>1991/06/27 01:17</td>
</tr>
<tr>
<td>1991/06/24 10:27</td>
<td>1991/06/24 10:27</td>
</tr>
</tbody>
</table>

If you hold the output in a comma-delimited or other alphanumeric output file, you can see that while the original field propagates only the numeric representation of the value, the converted field propagates the display options as well:

```plaintext
DEFINE FILE VIDEOTR2
DATE/I4 = HPART(TRANSDATE, 'YEAR', 'I4');
ALPHA_DATE/A16 = FPRINT(TRANSDATE,'HYYMDI', ALPHA_DATE);
END
TABLE FILE VIDEOTR2
PRINT TRANSDATE ALPHA_DATE/R
WHERE DATE EQ '1991'
ON TABLE HOLD FORMAT COMMA
END
```
FTOA: Converting a Number to Alphanumeric Format

How to:
Convert a Number to Alphanumeric Format

The FTOA function converts a number up to 16 digits long from numeric format to alphanumerical 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, or the name of the field that contains the number.

The HOLD file follows. The first field represents the original data, and the second field contains the converted values with display options:

"19910627024500000", "1991/06/27 02:45"
"19910620051500000", "1991/06/20 05:15"
"19910621071100000", "1991/06/21 07:11"
"19910621011000000", "1991/06/21 01:10"
"19910619071800000", "1991/06/19 07:18"
"19910619041100000", "1991/06/19 04:11"
"19910625011900000", "1991/06/25 01:19"
"19910624044300000", "1991/06/24 04:43"
"19910624020800000", "1991/06/24 02:08"
"19910625011700000", "1991/06/25 01:17"
"19910627011700000", "1991/06/27 01:17"
"19911117112800000", "1991/11/17 11:28"
"19910624102700000", "1991/06/24 10:27"
format

Alphanumeric

Is the format of the number to be converted enclosed in parentheses. Only floating point single-precision and double-precision formats are supported. Include any edit options that you want to appear in the output. The D (floating-point double-precision) format automatically supplies commas.

If you use a field name for this argument, specify the name without quotation marks or parentheses. If you specify a format, the format must be enclosed in single quotation marks and parentheses.

output

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length of this argument must be greater than the length of number and must account for edit options and a possible negative sign.

Example: Converting From Numeric to Alphanumeric Format

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

FTOA converts the GROSS field from floating point double-precision to alphanumeric format and stores the result in ALPHA_GROSS:

```
TABLE FILE EMPLOYEE
PRINT GROSS AND COMPUTE
ALPHA_GROSS/A15 = FTOA(GROSS, '(D12.2)', ALPHA_GROSS);
BY HIGHEST 1 PAY_DATE NOPRINT
BY LAST_NAME
WHERE (GROSS GT 800) AND (GROSS LT 2300);
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>GROSS</th>
<th>ALPHA_GROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>$1,815.00</td>
<td>1,815.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>$2,255.00</td>
<td>2,255.00</td>
</tr>
<tr>
<td>IRVING</td>
<td>$2,238.50</td>
<td>2,238.50</td>
</tr>
<tr>
<td>JONES</td>
<td>$1,540.00</td>
<td>1,540.00</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>$1,342.00</td>
<td>1,342.00</td>
</tr>
<tr>
<td>ROMANS</td>
<td>$1,760.00</td>
<td>1,760.00</td>
</tr>
<tr>
<td>SMITH</td>
<td>$1,100.00</td>
<td>1,100.00</td>
</tr>
<tr>
<td>STEVENS</td>
<td>$916.67</td>
<td>916.67</td>
</tr>
</tbody>
</table>
HEXBYT: Converting a Decimal Integer to a Character

How to:
Convert 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. For printable ASCII and EBCDIC characters and their integer equivalents see the Character Chart for ASCII and EBCDIC on page 34.

Syntax:  How to Convert a Decimal Integer to a Character

HEXBYT(decimal_value, output)

where:

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

- **output**
  - Alphanumeric
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
**Example: Converting a Decimal Integer to a Character**

HEXBYT converts LAST_INIT_CODE to its character equivalent and stores the result in LAST_INIT:

```
DEFINE FILE EMPLOYEE
PRINT LAST_NAME AND
COMPUTE LAST_INIT_CODE/I3 = BYTVAL(LAST_NAME, 'I3');
COMPUTE LAST_INIT/A1 = HEXBYT(LAST_INIT_CODE, LAST_INIT);
WHERE DEPARTMENT EQ 'MIS';
END
```

The output for an ASCII platform is:

```
LAST_NAME   LAST_INIT_CODE   LAST_INIT
-----------   ---------------  --------
SMITH        83 S
JONES        74 J
MCCOY        77 M
BLACKWOOD    66 B
GREENSPAN    71 G
CROSS        67 C
```

The output for an EBCDIC platform is:

```
LAST_NAME   LAST_INIT_CODE   LAST_INIT
-----------   ---------------  --------
SMITH        226 S
JONES        209 J
MCCOY        212 M
BLACKWOOD    194 B
GREENSPAN    199 G
CROSS        195 C
```

**Example: Inserting Braces for Mainframe**

HEXBYT converts the decimal integer 192 to its EBCDIC character equivalent, which is a left brace; and the decimal integer 208 to its character equivalent, which is a right brace. If the value of CURR_SAL is less than 12000, the value of LAST_NAME is enclosed in braces.

```
DEFINE FILE EMPLOYEE
BRACE/A17 = HEXBYT(192, 'A1') | LAST_NAME | HEXBYT(208, 'A1');
BNAME/A17 = IF CURR_SAL LT 12000 THEN BRACE ELSE LAST_NAME;
END
TABLE FILE EMPLOYEE
PRINT BNAME CURR_SAL BY EMP_ID
END
```
The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>BNAME</th>
<th>CURR_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>STEVENS</td>
<td>$11,000.00</td>
</tr>
<tr>
<td>112847612</td>
<td>SMITH</td>
<td>$13,200.00</td>
</tr>
<tr>
<td>117593129</td>
<td>JONES</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>119265415</td>
<td>(SMITH</td>
<td>$9,500.00</td>
</tr>
<tr>
<td>119329144</td>
<td>BANNING</td>
<td>$29,700.00</td>
</tr>
<tr>
<td>123764317</td>
<td>IRVING</td>
<td>$26,862.00</td>
</tr>
<tr>
<td>126724188</td>
<td>ROMANS</td>
<td>$21,120.00</td>
</tr>
<tr>
<td>219984371</td>
<td>MCCOY</td>
<td>$18,480.00</td>
</tr>
<tr>
<td>326179357</td>
<td>BLACKWOOD</td>
<td>$21,780.00</td>
</tr>
<tr>
<td>451123478</td>
<td>MCKNIGHT</td>
<td>$16,100.00</td>
</tr>
<tr>
<td>543729165</td>
<td>(GREENSPAN</td>
<td>$9,000.00</td>
</tr>
<tr>
<td>818692173</td>
<td>CROSS</td>
<td>$27,062.00</td>
</tr>
</tbody>
</table>

**ITONUM: Converting a Large Binary Integer to Double-Precision Format**

**How to:**

Convert a Large Binary Integer to Double-Precision Format

The ITONUM function converts a large binary integer in a non-FOCUS data source to double-precision format.

Some programming languages and some non-FOCUS data storage systems use large binary integer formats. However, large binary integers (more than 4 bytes in length) are not supported in the Master File so they require conversion to double-precision format.

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

**Syntax:**

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

`ITONUM(maxbytes, infield, output)`

where:

`maxbytes`

Numeric

Is the maximum number of bytes in the 8-byte binary input field that have significant numeric data, including the binary sign. Valid values are:

- 5 ignores the left-most 3 bytes.
- 6 ignores the left-most 2 bytes.
- 7 ignores the left-most byte.
**Example: Converting a Large Binary Integer to Double-Precision Format**

Suppose a binary number in an external file has the following COBOL format:

```
PIC 9(8)V9(4) COMP
```

It is defined in the EUROCAR Master File as a field named BINARYFLD. Its field formats are USAGE=A8 and ACTUAL=A8, since its length is greater than 4 bytes.

The following request converts the field to double-precision format:

```
DEFINE FILE EUROCAR
MYFLD/D14 = ITONUM(6, BINARYFLD, MYFLD);
END
TABLE FILE EUROCAR
PRINT MYFLD BY CAR
END
```

**ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format**

The ITOPACK function converts a large binary integer in a non-FOCUS data source to packed-decimal format.

Some programming languages and some non-FOCUS data storage systems use double-word binary integer formats. These are similar to the single-word binary integers used by FOCUS, but they allow larger numbers. However, large binary integers (more than 4 bytes in length) are not supported in the Master File so they require conversion to packed-decimal format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte packed-decimal field of up to 15 significant numeric positions (for example, P15 or P16.2).
**Limit:** For a field defined as 'PIC 9(15) COMP' or the equivalent (15 significant digits), the maximum number that can be converted is 167,744,242,712,576.

**Syntax:**

**How to Convert a Large Binary Integer to Packed-Decimal Format**

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

where:

- **maxbytes**
  - Numeric
  - Is the maximum number of bytes in the 8-byte binary input field that have significant numeric data, including the binary sign.
  - Valid values are:
    - 5 ignores the left-most 3 bytes (up to 11 significant positions).
    - 6 ignores the left-most 2 bytes (up to 14 significant positions).
    - 7 ignores the left-most byte (up to 15 significant positions).

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

- **output**
  - Numeric
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The format must be P\_n or P\_n.d.

**Example:**

**Converting a Large Binary Integer to Packed-Decimal Format**

Suppose a binary number in an external file has the following COBOL format:

P\_IC \_9\_8)\_V9\_4) \_COMP

It is defined in the EUROCAR Master File as a field named BINARYFLD. Its field formats are USAGE=A8 and ACTUAL=A8, since its length is greater than 4 bytes.

The following request converts the field to packed-decimal format:

\[
\text{DEFINE FILE EUROCAR}
\text{PACKFLD/P14.4 = \text{ITOPACK}(6, \text{BINARYFLD, \text{PACKFLD));}}
\text{END}
\text{TABLE FILE EUROCAR}
\text{PRINT PACKFLD BY CAR}
\text{END}
\]
ITOZ: Converting a Number to Zoned Format

How to:
Convert a Number to Zoned Format

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

Syntax: How to Convert a Number to Zoned Format

ITOZ(length, in_value, output)

where:

length
Integer
Is the length of in_value in bytes. The maximum number of bytes is 15. The last byte includes the sign.

in_value
Numeric
Is the number to be converted, or the field that contains the number. The number is truncated to an integer before it is converted.

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

Example: Converting a Number to Zoned Format

The following request creates an extract file containing employee IDs and salaries in zoned format for a COBOL program:

DEFINE FILE EMPLOYEE
ZONE_SAL/A8 = ITOZ(8, CURR_SAL, ZONE_SAL);
END

TABLE FILE EMPLOYEE
PRINT CURR_SAL ZONE_SAL BY EMP_ID
ON TABLE SAVE AS SALARIES
END
The resulting extract file is:

NUMBER OF RECORDS IN TABLE= 12 LINES= 12

[EBCDIC|ALPHANUMERIC] RECORD NAMED SALARIES

<table>
<thead>
<tr>
<th>FIELDNAME</th>
<th>ALIAS</th>
<th>FORMAT</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID</td>
<td>EID</td>
<td>A9</td>
<td>9</td>
</tr>
<tr>
<td>CURR_SAL</td>
<td>CSAL</td>
<td>D12.2M</td>
<td>12</td>
</tr>
<tr>
<td>ZONE_SAL</td>
<td></td>
<td>A8</td>
<td>8</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>29</td>
</tr>
</tbody>
</table>

DCB USED WITH FILE SALARIES IS DCB=(RECFM=FB,LRECL=00029,BLKSIZE=00580)

If you remove the SAVE command and run the request, the output for an EBCDIC platform follows. The left brace in EBCDIC is hexadecimal C0; this indicates a positive sign and a final digit of 0. The capital B in EBCDIC is hexadecimal C2; this indicates a positive sign and a final digit of 2.

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>URR_SAL</th>
<th>ZONE_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>$11,000.00</td>
<td>0001100{</td>
</tr>
<tr>
<td>112847612</td>
<td>$13,200.00</td>
<td>0001320{</td>
</tr>
<tr>
<td>117593129</td>
<td>$18,480.00</td>
<td>0001848{</td>
</tr>
<tr>
<td>119265415</td>
<td>$9,500.00</td>
<td>0000950{</td>
</tr>
<tr>
<td>119329144</td>
<td>$29,700.00</td>
<td>0002970{</td>
</tr>
<tr>
<td>123764317</td>
<td>$26,862.00</td>
<td>0002686B</td>
</tr>
<tr>
<td>126724188</td>
<td>$21,120.00</td>
<td>0002112{</td>
</tr>
<tr>
<td>219984371</td>
<td>$18,480.00</td>
<td>0001848{</td>
</tr>
<tr>
<td>326179357</td>
<td>$21,780.00</td>
<td>0002178{</td>
</tr>
<tr>
<td>451123478</td>
<td>$16,100.00</td>
<td>0001610{</td>
</tr>
<tr>
<td>543729165</td>
<td>$9,000.00</td>
<td>0000900{</td>
</tr>
<tr>
<td>818692173</td>
<td>$27,062.00</td>
<td>0002706B</td>
</tr>
</tbody>
</table>

**PCKOUT: Writing a Packed Number of Variable Length**

**How to:**
Write a Packed Number of Variable Length

The PCKOUT function writes a packed-decimal number of variable length to an extract file. When a request saves a packed number to an extract file, it typically writes it as an 8- or 16-byte field regardless of its format specification. With PCKOUT, you can vary the field's length between 1 to 16 bytes.

**Syntax:**

**How to Write a Packed Number of Variable Length**

PCKOUT(in_value, length, output)

where:

in_value
  Numeric
Is the input field that contains the values. It can be in packed, integer, single- or double-precision floating point format. If it is not in integer format, it is rounded to the nearest whole number.

\textit{length} \\
\text{Numeric} \\
Is the length of the output value, from 1 to 16 bytes.

\textit{output} \\
\text{Alphanumeric} \\
Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The function returns the field as alphanumeric although it contains packed data.

\textbf{Example: Writing a Packed Number of Variable Length}

PCKOUT converts the CURR_SAL field to a 5-byte packed field and stores the result in SHORT_SAL:

\begin{verbatim}
DEFINE FILE EMPLOYEE
SHORT_SAL/A5 = PCKOUT(CURR_SAL, 5, SHORT_SAL);
END
TABLE FILE EMPLOYEE
PRINT LAST_NAME SHORT_SAL HIRE_DATE
ON TABLE SAVE
END
\end{verbatim}

The resulting extract file is:

\begin{verbatim}
> NUMBER OF RECORDS IN TABLE= 12 LINES= 12
[EBCDIC|ALPHANUMERIC] RECORD NAMED SAVE
FIELDNAME ALIAS FORMAT LENGTH
LAST_NAME LN A15 15
SHORT_SAL A5 5
HIRE_DATE HDT I6YMD 6
TOTAL 26
DCB USED WITH FILE SAVE IS DCB=(RECFM=FB,LRECL=00026,BLKSIZE=000520)
\end{verbatim}

\textbf{PTOA: Converting a Packed-Decimal Number to Alphanumeric Format}

\textbf{How to:} \\
Convert a Packed-Decimal Number to Alphanumeric Format

The PTOA function converts a packed-decimal number from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can also add edit options to a number converted by PTOA.
When using PTOA to convert a number containing decimals to a character string, you must specify an alphanumeric format large enough to accommodate both the integer and decimal portions of the number. For example, a P12.2C format is converted to A14. If the output format is not large enough, the right-most characters are truncated.

**Syntax:** **How to Convert a Packed-Decimal Number to Alphanumeric Format**

```plaintext
PTOA(number, '(format)', output)
```

where:

- **number**
  - Numeric P (packed-decimal)
  - Is the number to be converted, or the name of the field that contains the number.

- **format**
  - Alphanumeric
  - Is the format of the number enclosed in both single quotation marks and parentheses.
  - Only packed-decimal format is supported. Include any edit options that you want to display in the output.
  - The format value does not require the same length or number of decimal places as the original field. If you change the number of decimal places, the result is rounded. If you make the length too short to hold the integer portion of the number, asterisks appear instead of the number.
  - If you use a field name for this argument, specify the name without quotation marks or parentheses. However, parentheses must be included around the format stored in this field. For example:
    ```plaintext
    FMT/A10 = '(P12.2C)';
    ```
  - You can then use this field as the format argument when using the function in your request:
    ```plaintext
    COMPUTE ALPHA_GROSS/A20 = PTOA(PGROSS, FMT, ALPHA_GROSS);
    ```

- **output**
  - Alphanumeric
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The length of this argument must be greater than the length of `number` and must account for edit options and a possible negative sign.
Example: Converting From Packed to Alphanumeric Format

PTOA is called twice to convert the PGROSS field from packed-decimal to alphanumeric format. The format specified in the first call to the function is stored in a virtual field named FMT. The format specified in the second call to the function does not include decimal places, so the value is rounded when it appears:

```
DEFINE FILE EMPLOYEE
   PGROSS/P18.2=GROSS;
   FMT/A10='(P14.2C)';
END
```

```
TABLE FILE EMPLOYEE PRINT PGROSS NOPRINT
   COMPUTE AGROSS/A17 = PTOA(PGROSS, FMT, AGROSS); AS ''
   COMPUTE BGROSS/A37 = '<- THIS AMOUNT IS' |
                         PTOA(PGROSS, '(P5C)', 'A6') |
                           ' WHEN ROUNDED'; AS '' IN +1
```

The output is:

- 2,475.00 <- THIS AMOUNT IS 2,475 WHEN ROUNDED
- 1,815.00 <- THIS AMOUNT IS 1,815 WHEN ROUNDED
- 2,255.00 <- THIS AMOUNT IS 2,255 WHEN ROUNDED
- 750.00 <- THIS AMOUNT IS 750 WHEN ROUNDED
- 2,238.50 <- THIS AMOUNT IS 2,239 WHEN ROUNDED
- 1,540.00 <- THIS AMOUNT IS 1,540 WHEN ROUNDED
- 1,540.00 <- THIS AMOUNT IS 1,540 WHEN ROUNDED
- 1,342.00 <- THIS AMOUNT IS 1,342 WHEN ROUNDED
- 1,760.00 <- THIS AMOUNT IS 1,760 WHEN ROUNDED
- 1,100.00 <- THIS AMOUNT IS 1,100 WHEN ROUNDED
- 791.67 <- THIS AMOUNT IS 792 WHEN ROUNDED
- 916.67 <- THIS AMOUNT IS 917 WHEN ROUNDED

UFMT: Converting an Alphanumeric String to Hexadecimal

How to:

Convert 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

UFMT(source_string, length, output)

where:

- **source_string**
  - Alphanumeric
  - Is the alphanumeric string to convert enclosed in single quotation marks, or the field that contains the string.

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

- **output**
  - Alphanumeric
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The format of output must be alphanumeric and its length must be twice that of length.

**Example:**

Converting an Alphanumeric String to Hexadecimal

UFMT converts each value in JOBCODE to its hexadecimal representation and stores the result in HEXCODE:

```plaintext
DEFINE FILE JOBFILE
HEXCODE/A6 = UFMT(JOBCODE, 3, HEXCODE);
END
TABLE FILE JOBFILE
PRINT JOBCODE HEXCODE
END
```

The output is:

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>HEXCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>C1F0F1</td>
</tr>
<tr>
<td>A02</td>
<td>C1F0F2</td>
</tr>
<tr>
<td>A07</td>
<td>C1F0F7</td>
</tr>
<tr>
<td>A12</td>
<td>C1F1F2</td>
</tr>
<tr>
<td>A14</td>
<td>C1F1F4</td>
</tr>
<tr>
<td>A15</td>
<td>C1F1F5</td>
</tr>
<tr>
<td>A16</td>
<td>C1F1F6</td>
</tr>
<tr>
<td>A17</td>
<td>C1F1F7</td>
</tr>
<tr>
<td>B01</td>
<td>C2F0F1</td>
</tr>
<tr>
<td>B02</td>
<td>C2F0F2</td>
</tr>
<tr>
<td>B03</td>
<td>C2F0F3</td>
</tr>
<tr>
<td>B04</td>
<td>C2F0F4</td>
</tr>
<tr>
<td>B14</td>
<td>C2F1F4</td>
</tr>
</tbody>
</table>
XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File

How to:
Store Packed Values in an Alphanumeric Field

The XTPACK function stores packed numbers with up to 31 significant digits in an alphanumeric field, retaining decimal data. This permits writing a short or long packed field of any length, 1 to 16 bytes, to an output file.

Syntax: How to Store Packed Values in an Alphanumeric Field

XTPACK(in_value, outlength, outdec, output)

where:

infield
Numeric
Is the packed value.

outlength
Numeric
Is the length of the alphanumeric field that will hold the converted packed field. Can be from 1 to 16.

outdec
Numeric
Is the number of decimal positions for output.

output
Alphanumeric
Is the name of the field to contain the result or the format of the field enclosed in single quotation marks.
**Example:** Writing a Long Packed Number to an Output File

The following request creates a long packed decimal field named LONGPCK. ALPHAPCK (format A13) is the result of applying XTPACK to the long packed field. PCT_INC, LONGPCK, and ALPHAPCK are then written to a SAVE file named XTOUT.

```
DEFINE FILE EMPLOYEE
LONGPCK/P25.2 = PCT_INC + 11111111111111111111;
ALPHAPCK/A13 = XTPACK(LONGPCK,13,2,'A13');
END
TABLE FILE EMPLOYEE
PRINT PCT_INC LONGPCK ALPHAPCK
WHERE PCT_INC GT 0
   ON TABLE SAVE AS XTOUT
END
```

The SAVE file has the following fields and formats:

```
ALPHANUMERIC RECORD NAMED XTOUT
FIELDNAME               ALIAS   FORMAT   LENGTH
PCT_INC                PI      F6.2      6
LONGPCK                PI      P25.2     25
ALPHAPCK               PI      A13       13
TOTAL                  PI      F6.2      6
SAVED...
```
Numeric functions perform calculations on numeric constants and fields.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function on page 43.

**Topics:**
- ABS: Calculating Absolute Value
- ASIS: Distinguishing Between a Blank and a Zero
- BAR: Producing a Bar Chart
- 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
- FMLINFO: Returning FOR Values
- FMLLIST: Returning an FML Tag List
- FMLFOR: Retrieving FML Tag Values
- FMLCAP: Retrieving FML Hierarchy Captions
- INT: Finding the Greatest Integer
- LOG: Calculating the Natural Logarithm
- MAX and MIN: Finding the Maximum or Minimum Value
- MIRR: Calculating the Modified Internal Return Rate
- NORMSDST: Calculating Standard Cumulative Normal Distribution
- NORMSINV: Calculating Inverse Cumulative Normal Distribution
- EXPN: Evaluating a Number in Scientific Notation
- PRDNOR and PRDUNI: Generating Reproducible Random Numbers
- PRDNOR and PRDUNI: Generating Random Numbers
- RDNORM and RDUNIF: Generating Random Numbers
- SQRT: Calculating the Square Root
- XIRR: Calculating the Modified Internal Return Rate (Periodic or Non-Periodic)
ABS: Calculating Absolute Value

**How to:**
Calculate Absolute Value

The ABS function returns the absolute value of a number.

**Syntax:**

**How to Calculate Absolute Value**

ABS(in_value)

where:

- **in_value**
  - Numeric
  - Is the value for which the absolute value is returned, the name of a field that contains the value, or an expression that returns the value. If you use an expression, use parentheses as needed to ensure the correct order of evaluation.

**Example:**

**Calculating Absolute Value**

The COMPUTE command creates the DIFF field, then ABS calculates the absolute value of DIFF:

```
TABLE FILE SALES
PRINT UNIT_SOLD AND DELIVER_AMT AND
COMPUTE DIFF/I5 = DELIVER_AMT - UNIT_SOLD; AND
COMPUTE ABS_DIFF/I5 = ABS(DIFF);
BY PROD_CODE
WHERE DATE LE '1017';
END
```

The output is:

<table>
<thead>
<tr>
<th>PROD_CODE</th>
<th>UNIT_SOLD</th>
<th>DELIVER_AMT</th>
<th>DIFF</th>
<th>ABS_DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>B10</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B17</td>
<td>20</td>
<td>40</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>B20</td>
<td>15</td>
<td>30</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>C17</td>
<td>12</td>
<td>10</td>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>D12</td>
<td>20</td>
<td>30</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>E1</td>
<td>30</td>
<td>25</td>
<td>-5</td>
<td>5</td>
</tr>
<tr>
<td>E3</td>
<td>35</td>
<td>25</td>
<td>-10</td>
<td>10</td>
</tr>
</tbody>
</table>
**ASIS: Distinguishing Between a Blank and a Zero**

The ASIS function distinguishes between a blank and a zero in Dialogue Manager. It differentiates between a numeric string constant or variable defined as a numeric string, and a field defined simply as numeric.

For details on ASIS, see *ASIS: Distinguishing Between Space and Zero* on page 69.

**BAR: Producing a Bar Chart**

**How to:**

Produce a Bar Chart

The BAR function produces a horizontal bar chart using repeating characters to form each bar. Optionally, you can create a scale to clarify the meaning of a bar chart by replacing the title of the column containing the bar with a scale.

**Syntax:**

**How to Produce a Bar Chart**

BAR(barlength, infield, maxvalue, 'char', output)

where:

**barlength**

Numeric

Is the maximum length of the bar, in characters. If this value is less than or equal to 0, the function does not return a bar.

**infield**

Numeric

Is the data field plotted as a bar chart.

**maxvalue**

Numeric

Is the maximum value of a bar. This value must be greater than the maximum value stored in **infield**. If **infield** is larger than **maxvalue**, the function uses **maxvalue** and returns a bar of maximum length.

**'char'**

Alphanumeric

Is the repeating character that creates the bars enclosed in single quotation marks. If you specify more than one character, only the first character is used.
output

Alphanumeric

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The output field must be large enough to contain a bar of maximum length as defined by `barlength`.

Example: Producing a Bar Chart

BAR creates a bar chart for the CURR_SAL field, and stores the output in SAL_BAR. The bar created can be no longer than 30 characters long, and the value it represents can be no greater than 30,000.

```plaintext
TABLE FILE EMPLOYEE
PRINT CURR_SAL AND COMPUTE
SAL_BAR/A30 = BAR(30, CURR_SAL, 30000, '=', SAL_BAR);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION';
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>CURR_SAL</th>
<th>SAL_BAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$29,700.00</td>
<td>===================================================</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$26,862.00</td>
<td>===================================================</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$16,100.00</td>
<td>===================================================</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$21,120.00</td>
<td>===================================================</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$9,500.00</td>
<td>===================================================</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$11,000.00</td>
<td>===================================================</td>
</tr>
</tbody>
</table>
**Example:** Creating a Bar Chart With a Scale

BAR creates a bar chart for the CURR_SAL field. The request then replaces the field name SAL_BAR with a scale using the AS phrase.

To run this request on a platform for which the default font is proportional, use a non-proportional font or issue SET STYLE=OFF.

```
TABLE FILE EMPLOYEE
HEADING
"CURRENT SALARIES OF EMPLOYEES IN PRODUCTION DEPARTMENT"
"GRAPHED IN THOUSANDS OF DOLLARS"
""
PRINT CURR_SAL AS 'CURRENT SALARY'
AND COMPUTE
  SAL_BAR/A30 = BAR(30, CURR_SAL, 30000, '='='', SAL_BAR);
AS
  '  5 10 15 20 25 30,------------------------'
BY LAST_NAME AS 'LAST NAME'
BY FIRST_NAME AS 'FIRST NAME'
WHERE DEPARTMENT EQ 'PRODUCTION';
ON TABLE SET PAGE-NUM OFFEND
```

The output is:

```
CURRENT SALARIES OF EMPLOYEES IN PRODUCTION DEPARTMENT
GRAPHED IN THOUSANDS OF DOLLARS

  5 10 15 20 25 30

<table>
<thead>
<tr>
<th>LAST NAME</th>
<th>FIRST NAME</th>
<th>CURRENT SALARY</th>
<th>--------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>$29,700.00</td>
<td>===============</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>$26,862.00</td>
<td>==============</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>$16,100.00</td>
<td>===============</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>$21,120.00</td>
<td>===============</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>$9,500.00</td>
<td>==============</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>$11,000.00</td>
<td>==============</td>
</tr>
</tbody>
</table>
```

**CHKPCK: Validating a Packed Field**

**How to:** Validate 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) or the MODIFY FIXFORM command defines the field as alphanumeric, not packed. This does not change the field data, which remains packed, but it enables the request to read the data without a data exception.

2. Call CHKPCK to examine the field. The function returns the output to a field defined as packed. If the value it examines is a valid packed number, the function returns the value; if the value is not packed, the function returns an error code.

**Syntax: How to Validate a Packed Field**

CHKPCK(length, in_value, error, output)

where:

*length*

Numeric

Is the length of the packed field. It can be between 1 and 16 bytes.

*infield*

Alphanumeric

Is the name of the packed field or the value to be verified as packed decimal. Is the. The value must be described as alphanumeric, not packed.

*error*

Numeric

Is the error code that the function returns if a value is not packed. Choose an error code outside the range of data. The error code is first truncated to an integer, then converted to packed format. However, it may appear on a report with a decimal point depending on the output format.

*output*

Packed-decimal

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.
Example: Validating Packed Data

1. Prepare a data source that includes invalid packed data. The following example creates TESTPACK, which contains the PACK_SAL field. PACK_SAL is defined as alphanumeric but actually contains packed data. The invalid packed data is stored as AAA.

```assembler
DEFINE FILE EMPLOYEE
PACK_SAL/A8 = IF EMP_ID CONTAINS '123'
    THEN 'AAA' ELSE PCKOUT(CURR_SAL, 8, 'A8');
END

TABLE FILE EMPLOYEE
PRINT DEPARTMENT PACK_SAL BY EMP_ID
ON TABLE SAVE AS TESTPACK
END

The output is:

> NUMBER OF RECORDS IN TABLE= 12 LINES= 12
[EBBCDIC|ALPHANUMERIC] RECORD NAMED TESTPACK
FIELDNAME ALIAS FORMAT LENGTH
EMP_ID EID A9 9
DEPARTMENT DPT A10 10
PACK_SAL A8 8
TOTAL 27
[DCB USED WITH FILE TESTPACK IS DCB=(RECFM=FB,LRECL=00027,BLKSIZE=00540)]
SAVED... >
```

2. Create a Master File for the TESTPACK data source. Define the PACK_SAL field as alphanumeric in the USAGE and ACTUAL attributes.

```assembler
FILE = TESTPACK, SUFFIX = FIX
FIELD = EMP_ID, ALIAS = EID, USAGE = A9, ACTUAL = A9,
FIELD = DEPARTMENT, ALIAS = DPT, USAGE = A10, ACTUAL = A10,
FIELD = PACK_SAL, ALIAS = PS, USAGE = A8, ACTUAL = A8,
```

3. Create a request that uses CHKPCK to validate the values in the PACK_SAL field, and store the result in the GOOD_PACK field. Values not in packed format return the error code -999. Values in packed format appear accurately.

```assembler
DEFINE FILE TESTPACK
GOOD_PACK/P8CM = CHKPCK(8, PACK_SAL, -999, GOOD_PACK);
END

TABLE FILE TESTPACK
PRINT DEPARTMENT GOOD_PACK BY EMP_ID
END
```
The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>DEPARTMENT</th>
<th>GOOD_PACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>PRODUCTION</td>
<td>$11,000</td>
</tr>
<tr>
<td>112847612</td>
<td>MIS</td>
<td>$13,200</td>
</tr>
<tr>
<td>117593129</td>
<td>MIS</td>
<td>$18,480</td>
</tr>
<tr>
<td>119265415</td>
<td>PRODUCTION</td>
<td>$9,500</td>
</tr>
<tr>
<td>119329144</td>
<td>PRODUCTION</td>
<td>$29,700</td>
</tr>
<tr>
<td>123764317</td>
<td>PRODUCTION</td>
<td>-$999</td>
</tr>
<tr>
<td>126724188</td>
<td>PRODUCTION</td>
<td>$21,120</td>
</tr>
<tr>
<td>219984371</td>
<td>MIS</td>
<td>$18,480</td>
</tr>
<tr>
<td>326179357</td>
<td>MIS</td>
<td>$21,780</td>
</tr>
<tr>
<td>451123478</td>
<td>PRODUCTION</td>
<td>-$999</td>
</tr>
<tr>
<td>543729165</td>
<td>MIS</td>
<td>$9,000</td>
</tr>
<tr>
<td>818692173</td>
<td>MIS</td>
<td>$27,062</td>
</tr>
</tbody>
</table>

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

**How to:**
Calculate the Remainder From a Division

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

The functions use the following formula.

\[
remainder = dividend - \text{INT}(dividend/divisor) \times divisor
\]

- **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 359.

**Syntax:**

**How to Calculate the Remainder From a Division**

\[
\text{function}(\text{dividend}, \text{divisor}, \text{output})
\]

where:

function

Is one of the following:

- **DMOD** returns the remainder as a decimal number.
- **FMOD** returns the remainder as a floating-point number.
- **IMOD** returns the remainder as an integer.
**dividend**

Numeric

Is the number being divided.

**divisor**

Numeric

Is the number dividing the dividend.

**output**

Numeric

Is the result whose format is determined by the function used. Can be the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

**Example: Calculating the Remainder From a Division**

IMOD divides ACCTNUMBER by 1000 and returns the remainder to LAST3_ACCT:

```
TABLE FILE EMPLOYEE
PRINT ACCTNUMBER AND COMPUTE
LAST3_ACCT/I3L = IMOD(ACCTNUMBER, 1000, LAST3_ACCT);
BY LAST_NAME BY FIRST_NAME
WHERE (ACCTNUMBER NE 000000000) AND (DEPARTMENT EQ 'MIS');
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ACCTNUMBER</th>
<th>LAST3_ACCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>122850108</td>
<td>108</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>163800144</td>
<td>144</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>150150302</td>
<td>302</td>
</tr>
<tr>
<td>MCCOY</td>
<td>DIANE</td>
<td>040950036</td>
<td>036</td>
</tr>
<tr>
<td>SMITH</td>
<td>JOHN</td>
<td>109200096</td>
<td>096</td>
</tr>
<tr>
<td></td>
<td>MARY</td>
<td>027300024</td>
<td>024</td>
</tr>
</tbody>
</table>

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

**How to:**

Raise e to the Nth Power

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

\[ \text{EXP}(\text{power}, \text{output}) \]

where:

- **power**
  - Numeric
  - Is the power to which "e" is raised.

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

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

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

- `SET &POW = '3';`
- `SET &RESULT = EXP(&POW, 'D15.3') + 0.5;`
- `TYPE E TO THE &POW POWER IS APPROXIMATELY &RESULT`

The output is:

E TO THE 3 POWER IS APPROXIMATELY 20

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

**How to:**

Evaluate a Number in Scientific Notation

The EXPN function evaluates a number expressed in scientific notation.

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

\[ \text{EXPN}(n.nn \ \{E|D\} \ (+|-) \ p) \]
where:

\[ n.nn \]

Numeric

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

\[ E, D \]

Denotes scientific notation. E and D are interchangeable.

\[ +, - \]

Indicates if \( p \) is positive or negative.

\[ p \]

Integer

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

**Note:** EXPN does not use an output argument. The format of the result is floating-point double precision.

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

EXPN evaluates SCI_DATA.

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

For 1.03E+2, the result is 103.

**FMLINFO: Returning FOR Values**

**How to:**

Retain FOR Values in an FML Request

The FMLINFO function returns the FOR value associated with each row in an FML report. With FMLINFO, you can use the appropriate FOR value in a COMPUTE command to do drill-downs and sign changes for each row in the report, even when the row is a summary row created using an OR list or a Financial Modeling Language (FML) Hierarchy ADD command.

**Note:** You can use the SET parameter FORMULTIPLE=ON to enable an incoming record to be used on more than one line in an FML report.
**Syntax:**  
How to Retain FOR Values in an FML Request

FMLINFO('FORVALUE', output)

where:

'FORVALUE'

Alphanumeric

Returns the FOR value associated with each row in an FML report. If the FML row was generated as a sum of data records using the OR phrase, FMLINFO returns the first FOR value specified in the list of values. If the OR phrase was generated by an FML Hierarchy ADD command, FMLINFO returns the FOR value associated with the parent specified in the ADD command.

output

Alphanumeric

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

**Example:**  
Retrieving FOR Values for FML Hierarchy Rows

The following request creates a field called PRINT_AMT that is the negative of the NAT_AMOUNT field for account numbers less than 2500 in the CENTSYSF data source. The CENTGL data source contains the hierarchy information for CENTSYSF. Therefore, CENTGL is joined to CENTSYSF for the request:

```
SET FORMULTIPLE = ON
JOIN SYS_ACCOUNT IN CENTGL TO ALL SYS_ACCOUNT IN CENTSYSF
TABLE FILE CENTGL
SUM NAT_AMOUNT/D10 IN 30
COMPUTE PRINT_AMT/D10 = IF FMLINFO('FORVALUE', 'A7') LT '2500'
  THEN 0-NAT_AMOUNT ELSE NAT_AMOUNT;
COMPUTE FORV/A4 = FMLINFO('FORVALUE', 'A4');
COMPUTE ACTION/A9 = IF FORV LT '2500'
  THEN 'CHANGED' ELSE 'UNCHANGED';
FOR GL_ACCOUNT
2000 WITH CHILDREN 2 ADD AS CAPTION
END
```
**Note:** The parent value specified in the WITH CHILDREN ADD command (2000) is returned for the first row on the report. Each subsequent row is also a consolidated subsection of the hierarchy with a parent value that is returned by FMLINFO:

<table>
<thead>
<tr>
<th>Month</th>
<th>Actual</th>
<th>PRINT_AMT</th>
<th>FORV</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gross Margin</td>
<td>-25,639,223</td>
<td>25,639,223</td>
<td>2000</td>
<td>CHANGED</td>
</tr>
<tr>
<td>Sales Revenue</td>
<td>-62,362,490</td>
<td>62,362,490</td>
<td>2100</td>
<td>CHANGED</td>
</tr>
<tr>
<td>Retail Sales</td>
<td>-49,355,184</td>
<td>49,355,184</td>
<td>2200</td>
<td>CHANGED</td>
</tr>
<tr>
<td>Mail Order Sales</td>
<td>-6,899,416</td>
<td>6,899,416</td>
<td>2300</td>
<td>CHANGED</td>
</tr>
<tr>
<td>Internet Sales</td>
<td>-6,107,890</td>
<td>6,107,890</td>
<td>2400</td>
<td>CHANGED</td>
</tr>
<tr>
<td>Cost Of Goods Sold</td>
<td>36,723,267</td>
<td>36,723,267</td>
<td>2500</td>
<td>UNCHANGED</td>
</tr>
<tr>
<td>Variable Material Costs</td>
<td>27,438,625</td>
<td>27,438,625</td>
<td>2600</td>
<td>UNCHANGED</td>
</tr>
<tr>
<td>Direct Labor</td>
<td>6,176,900</td>
<td>6,176,900</td>
<td>2700</td>
<td>UNCHANGED</td>
</tr>
<tr>
<td>Fixed Costs</td>
<td>3,107,742</td>
<td>3,107,742</td>
<td>2800</td>
<td>UNCHANGED</td>
</tr>
</tbody>
</table>

**Example:** Using FMLINFO With an OR Phrase

The FOR value printed for the summary line is 1010, but FMLINFO returns the first value specified in the OR list, 1030:

```sql
SET FORMULTIPLE = ON
TABLE FILE LEDGER
SUM AMOUNT
COMPUTE RETURNEDFOR/A8 = FMLINFO('FORVALUE', 'A8');
FOR ACCOUNT
1010     OVER
1020     OVER
1030     OVER
BAR      OVER
1030 OR 1020 OR 1010
END
```

The output is:

<table>
<thead>
<tr>
<th>AMOUNT</th>
<th>RETURNEDFOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>8,784</td>
</tr>
<tr>
<td>1020</td>
<td>4,494</td>
</tr>
<tr>
<td>1030</td>
<td>7,961</td>
</tr>
<tr>
<td>______</td>
<td>_______</td>
</tr>
<tr>
<td>1010</td>
<td>21,239</td>
</tr>
</tbody>
</table>

**FMLLIST: Returning an FML Tag List**

**How to:**

Retrieve an FML Tag List

FMLLIST returns a string containing the complete tag list for each row in an FML request. If a row has a single tag value, that value is returned.
The FMLLIST function is supported for COMPUTE but not for DEFINE. Attempts to use it in a DEFINE result in blank values.

**Syntax:**

**How to Retrieve an FML Tag List**

FMLLIST('A4096V')

where:

'A4096V'

Is the required argument.

**Example:**

**Retrieving an FML Tag List With FMLLIST**

SET FORMULTIPLE=ON
TABLE FILE LEDGER
HEADING
"TEST OF FMLLIST"
" "
SUM AMOUNT
COMPUTE LIST1/A36 = FMLLIST('A4096V');
FOR ACCOUNT
'1010' OVER
'1020' OVER
'1030' OVER
BAR OVER
'1030' OR '1020' OR '1010'
END

The output is:

<table>
<thead>
<tr>
<th>TEST OF FMLLIST</th>
<th>AMOUNT</th>
<th>LIST1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>8,784</td>
<td>1010</td>
</tr>
<tr>
<td>1020</td>
<td>4,494</td>
<td>1020</td>
</tr>
<tr>
<td>1030</td>
<td>7,961</td>
<td>1030</td>
</tr>
<tr>
<td>1010 OR 1020 OR 1030</td>
<td>---------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>1010</td>
<td>21,239</td>
<td>1010 OR 1020 OR 1030</td>
</tr>
</tbody>
</table>
FMLFOR: Retrieving FML Tag Values

FMLFOR retrieves the tag value associated with each row in an FML request. If the FML row was generated as a sum of data records using the OR phrase, FMLFOR returns the first value specified in the list. If the OR phrase was generated by an FML Hierarchy ADD command, FMLFOR returns the tag value associated with the parent specified in the ADD command.

The FMLFOR function is supported for COMPUTE but not for DEFINE. Attempts to use it in a DEFINE result in blank values.

Syntax: How to Retrieve FML Tag Values

FMLFOR(output)

where:

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

Example: Retrieving FML Tag Values With FMLFOR

SET FORMULTIPLE = ON
TABLE FILE LEDGER
SUM AMOUNT
COMPUTE RETURNEDFOR/A8 = FMLFOR('A8');
FOR ACCOUNT
  1010                  OVER
  1020                  OVER
  1030                  OVER
  BAR                   OVER
  1030 OR 1020 OR 1010
END

The output is:

<table>
<thead>
<tr>
<th></th>
<th>RETURNEDFOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>8,784</td>
</tr>
<tr>
<td>1020</td>
<td>4,494</td>
</tr>
<tr>
<td>1030</td>
<td>7,961</td>
</tr>
<tr>
<td>1010</td>
<td>21,239</td>
</tr>
</tbody>
</table>
FMLCAP: Retrieving FML Hierarchy Captions

How to:
Retrieve Captions in an FML Request Using the FMLCAP Function

The FMLCAP function returns the caption value for each row in an FML hierarchy request. In order to retrieve caption values, the Master File must define an FML hierarchy and the request must use the GET CHILDREN, ADD, or WITH CHILDREN option to retrieve hierarchy data. If the FOR field in the request does not have a caption field defined, FMLCAP returns a blank string.

FMLCAP is supported for COMPUTE but is not recommended for use with DEFINE.

Syntax: How to Retrieve Captions in an FML Request Using the FMLCAP Function

FMLCAP (fieldname | 'format')

where:

fieldname
  Is the name of the caption field.

'format'
  Is the format of the caption field enclosed in single quotation marks.

Example: Retrieving FML Hierarchy Captions Using FMLCAP

The following request retrieves and aggregates the FML hierarchy that starts with the parent value 2000. FMLCAP retrieves the captions, while the actual account numbers appear as the FOR values.

SET FORMULTIPLE = ON
TABLE FILE CENTSTMT
SUM ACTUAL_AMT
COMPUTE CAP1/A30 = FMLCAP (GL_ACCOUNT_CAPTION);  
FOR GL_ACCOUNT
2000 WITH CHILDREN 2 ADD
END
The output is:

<table>
<thead>
<tr>
<th>Actual</th>
<th>CAP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>313,611,852.</td>
</tr>
<tr>
<td>2100</td>
<td>187,087,470.</td>
</tr>
<tr>
<td>2200</td>
<td>98,710,368.</td>
</tr>
<tr>
<td>2300</td>
<td>13,798,832.</td>
</tr>
<tr>
<td>2400</td>
<td>12,215,780.</td>
</tr>
<tr>
<td>2500</td>
<td>100,885,159.</td>
</tr>
<tr>
<td>2600</td>
<td>54,877,250.</td>
</tr>
<tr>
<td>2700</td>
<td>6,176,900.</td>
</tr>
<tr>
<td>2800</td>
<td>3,107,742.</td>
</tr>
</tbody>
</table>

**INT: Finding the Greatest Integer**

**How to:**

Find the Greatest Integer

The INT function returns the integer component of a number.

**Syntax:**

**How to Find the Greatest Integer**

```
INT(in_value)
```

where:

- `in_value`
  - Numeric

Is the value for which the integer component is returned, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

**Example:**

Finding the Greatest Integer

INT finds the greatest integer in the DED_AMT field and stores it in INT_DED_AMT:

```
TABLE FILE EMPLOYEE
SUM DED_AMT AND COMPUTE
INT_DED_AMT/I9 = INT(DED_AMT);
BY LAST_NAME BY FIRST_NAME
WHERE (DEPARTMENT EQ 'MIS') AND (PAY_DATE EQ 820730);
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>DED_AMT</th>
<th>INT_DED_AMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>$1,261.40</td>
<td>1261</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>$1,668.69</td>
<td>1668</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>$127.50</td>
<td>127</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>$725.34</td>
<td>725</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>$334.10</td>
<td>334</td>
</tr>
</tbody>
</table>

LOG: Calculating the Natural Logarithm

How to:
Calculate the Natural Logarithm

The LOG function returns the natural logarithm of a number.

Syntax: How to Calculate the Natural Logarithm

LOG(in_value)

where:

in_value

Numeric

Is the value for which the natural logarithm is calculated, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If in_value is less than or equal to 0, LOG returns 0.

Example: Calculating the Natural Logarithm

LOG calculates the logarithm of the CURR_SAL field:

TABLE FILE EMPLOYEE
PRINT CURR_SAL AND COMPUTE
LOG_CURR_SAL/D12.2 = LOG (CURR_SAL) ;
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'PRODUCTION' ;
END
MAX and MIN: Finding the Maximum or Minimum Value

**How to:**

Find the Maximum or Minimum Value

The MAX and MIN functions return the maximum or minimum value, respectively, from a list of values.

**Syntax:**

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

where:

MAX

Returns the maximum value.

MIN

Returns the minimum value.

value1, value2

Numeric

Are the values for which the maximum or minimum value is returned, the name of a field that contains the values, or an expression that returns the values. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

**Example:**

Determining the Minimum Value

MIN returns either the value of the ED_HRS field or the constant 30, whichever is lower:

```
TABLE FILE EMPLOYEE
PRINT ED_HRS AND COMPUTE
MIN_EDHRS_30/D12.2 = MIN(ED_HRS, 30);
BY LAST_NAME BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
END
```
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ED_HRS</th>
<th>MIN_EDHRS_30</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>75.00</td>
<td>30.00</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>45.00</td>
<td>30.00</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>25.00</td>
<td>25.00</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>50.00</td>
<td>30.00</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>36.00</td>
<td>30.00</td>
</tr>
</tbody>
</table>

**MIRR: Calculating the Modified Internal Return Rate**

**How to:**
Calculate the Modified Internal Rate of Return

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

The MIRR function calculates the modified internal rate of return for a series of periodic cash flows.

**Syntax:**

**How to Calculate the Modified Internal Rate of Return**

```plaintext
TABLE FILE ...
{PRINT|SUM} field ...
COMPUTE rrate/fmt = MIRR(cashflow, finrate, reinvrate, output);
WITHIN {sort_field|TABLE}
```

where:

- `field ...`
  Are fields that appear in the report output.
- `rrate`
  Is the field that contains the calculated return rate.
- `fmt`
  Is the format of the return rate. The data type must be D.
- `cashflow`
  Is a numeric field. Each value represents either a payment (negative value) or income (positive value) for one period. The values must be in the correct sequence in order for the sequence of cash flows to be calculated correctly. The dates corresponding to each cash flow should be equally spaced and sorted in chronological order. The calculation requires at least one negative value and one positive value in the `cashflow` field. If the values are all positive or all negative, a zero result is returned.
**finrate**

Is a finance rate for negative cash flows. This value must be expressed as a non-negative decimal fraction between 0 and 1. It must be constant within each sort group for which a return rate is calculated, but it can change between sort groups.

**reinvrate**

Is the reinvestment rate for positive cash flows. This value must be expressed as a non-negative decimal fraction between 0 and 1. It must be constant within each sort group but can change between sort groups. It must be constant within each sort group for which a return rate is calculated, but it can change between sort groups.

**output**

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

**sort_field**

Is a field that sorts the report output and groups it into subsets of rows on which the function can be calculated separately. To calculate the function using every row of the report output, use the WITHIN TABLE phrase. A WITHIN phrase is required.

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

- This function is only supported in a COMPUTE command with the WITHIN phrase.
- The cash flow field must contain at least one negative value and one positive value.
- Dates must be equally spaced.
- Missing cash flows or dates are not supported.

**Example:** Calculating the Modified Internal Rate of Return

The following request calculates modified internal return rates for categories of products. It assumes a finance charge of ten percent and a reinvestment rate of ten percent. The request is sorted by date so that the correct cash flows are calculated. The rate returned by the function is multiplied by 100 in order to express it as a percent rather than a decimal value. Note that the format includes the % character. This causes a percent symbol to display, but it does not calculate a percent.
In order to create one cash flow value per date, the values are summed. NEWDOLL is defined in order to create negative values in each category as required by the function:

```
DEFINE FILE GGSALES
  SDATE/YYM = DATE;
  SYEAR/Y = SDATE;
  NEWDOLL/D12.2 = IF DATE LT '19970401' THEN -1 * DOLLARS ELSE DOLLARS;
END
```

```
TABLE FILE GGSALES
  SUM NEWDOLL
  COMPUTE RRATE/D7.2% = MIRR(NEWDOLL, .1, .1, RRATE) * 100;
  WITHIN CATEGORY
    BY CATEGORY
    BY SDATE
    WHERE SYEAR EQ 97
END
```

A separate rate is calculated for each category because of the WITHIN CATEGORY phrase. A portion of the output is shown:

<table>
<thead>
<tr>
<th>Category</th>
<th>SDATE</th>
<th>NEWDOLL</th>
<th>RRATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997/01</td>
<td>-801,123.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/02</td>
<td>-682,340.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/03</td>
<td>-765,078.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/04</td>
<td>691,274.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/05</td>
<td>720,444.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/06</td>
<td>742,457.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/07</td>
<td>747,253.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/08</td>
<td>655,896.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/09</td>
<td>730,317.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/10</td>
<td>724,412.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/11</td>
<td>620,264.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>1997/12</td>
<td>762,328.00</td>
<td>15.11%</td>
<td></td>
</tr>
<tr>
<td>Food</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997/01</td>
<td>-672,727.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/02</td>
<td>-699,073.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/03</td>
<td>-642,802.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/04</td>
<td>718,514.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/05</td>
<td>660,740.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/06</td>
<td>734,705.00</td>
<td>16.24%</td>
<td></td>
</tr>
<tr>
<td>1997/07</td>
<td>760,586.00</td>
<td>16.24%</td>
<td></td>
</tr>
</tbody>
</table>
To calculate one modified internal return rate for all of the report data, use the WITHIN TABLE phrase. In this case, the data does not have to be sorted by CATEGORY:

```
DEFINE FILE GGSALES
 SDATE/YYM = DATE;
 SYEAR/Y = SDATE;
 NEWDOLL/D12.2 = IF DATE LT '19970401' THEN -1 * DOLLARS ELSE DOLLARS;
END

TABLE FILE GGSALES
 SUM NEWDOLL
 COMPUTE RRATE/D7.2% = MIRR(NEWDOLL, .1, .1, RRATE) * 100;
 WITHIN TABLE
 BY SDATE
 WHERE SYEAR EQ 97
END
```

The output is:

<table>
<thead>
<tr>
<th>SDATE</th>
<th>NEWDOLL</th>
<th>RRATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997/01</td>
<td>-1,864,129.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/02</td>
<td>-1,861,639.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/03</td>
<td>-1,874,439.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/04</td>
<td>1,829,838.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/05</td>
<td>1,899,494.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/06</td>
<td>1,932,630.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/07</td>
<td>2,005,402.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/08</td>
<td>1,838,863.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/09</td>
<td>1,893,944.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/10</td>
<td>1,933,705.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/11</td>
<td>1,865,982.00</td>
<td>15.92%</td>
</tr>
<tr>
<td>1997/12</td>
<td>2,053,923.00</td>
<td>15.92%</td>
</tr>
</tbody>
</table>

**NORMSDST: Calculating Standard Cumulative Normal Distribution**

**How to:**
Calculate the Cumulative Standard Normal Distribution Function

**Reference:**
Characteristics of the Normal Distribution

The NORMSDST function performs calculations on a standard normal distribution curve, calculating the percentage of data values that are less than or equal to a normalized value. A normalized value is a point on the X-axis of a standard normal distribution curve in standard deviations from the mean. This is useful for determining percentiles in normally distributed data.
The NORMSINV function is the inverse of NORMSDST. For information about NORMSINV, see *NORMSINV: Calculating Inverse Cumulative Normal Distribution* on page 368.

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:

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

The following diagram illustrates the results of the NORMSDST and NORMSINV functions.
Reference: Characteristics of the Normal Distribution

Many common measurements are normally distributed. A plot of normally distributed data values approximates a bell-shaped curve. The two measures required to describe any normal distribution are the mean and the standard deviation:

- The mean is the point at the center of the curve.
- The standard deviation describes the spread of the curve. It is the distance from the mean to the point of inflection (where the curve changes direction).

Syntax: How to Calculate the Cumulative Standard Normal Distribution Function

NORMSDST(value, 'D8');

where:

value

Is a normalized value.

D8

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

Example: Using the NORMSDST Function

NORMSDST calculates the Z value and finds its percentile:

DEFINE FILE GGPRODS
- Convert SIZE FIELD TO DOUBLE PRECISION
X/D12.5 = SIZE;
END
TABLE FILE GGPRODS
SUM X NOPRINT CNT.X NOPRINT
- Calculate mean and standard deviation
COMPUTE NUM/D12.5 = CNT.X; NOPRINT
COMPUTE MEAN/D12.5 = AVE.X; NOPRINT
COMPUTE VARIANCE/D12.5 = ((NUM*ASQ.X) - (X*X/NUM))/(NUM-1); NOPRINT
COMPUTE STDEV/D12.5 = SQRT(VARIANCE); NOPRINT
PRINT SIZE X NOPRINT
- Compute normalized values and use as input to NORMSDST function
COMPUTE Z/D12.5 = (X - MEAN)/STDEV;
COMPUTE NORMSD/D12.5 = NORMSDST(Z, 'D8');
BY PRODUCT_ID NOPRINT
END
The output is:

<table>
<thead>
<tr>
<th>Size</th>
<th>Z</th>
<th>NORMSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
</tr>
<tr>
<td>20</td>
<td>.65678</td>
<td>.74434</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>.91721</td>
</tr>
<tr>
<td>16</td>
<td>-.07298</td>
<td>.47091</td>
</tr>
<tr>
<td>12</td>
<td>-.80273</td>
<td>.21106</td>
</tr>
<tr>
<td>8</td>
<td>-1.53249</td>
<td>.06270</td>
</tr>
</tbody>
</table>

NORMSINV: Calculating Inverse Cumulative Normal Distribution

**How to:**

Calculate the Inverse Cumulative Standard Normal Distribution Function

The NORMSINV function performs calculations on a standard normal distribution curve, finding the normalized value that forms the upper boundary of a percentile in a standard normal distribution curve. This is the inverse of NORMSDST. For information about NORMSDST, see [NORMSDST: Calculating Standard Cumulative Normal Distribution](page365) on page 365.

The results of NORMSINV are returned as double-precision and are accurate to 6 significant digits.

**Syntax:**

How to Calculate the Inverse Cumulative Standard Normal Distribution Function

NORMSINV(value, 'D8');

where:

- **value**
  
  Is a number between 0 and 1 (which represents a percentile in a standard normal distribution).

- **D8**

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

NORMSDST finds the percentile for the Z field. NORMSINV then returns this percentile to a normalized value:

```
DEFINE FILE GGPRODS
  -* CONVERT SIZE FIELD TO DOUBLE PRECISION
  X/D12.5 = SIZE;
END

TABLE FILE GGPRODS
  SUM X NOPRINT CNT.X NOPRINT
  -* CALCULATE MEAN AND STANDARD DEVIATION
  COMPUTE NUM/D12.5 = CNT.X; NOPRINT
  COMPUTE MEAN/D12.5 = AVE.X; NOPRINT
  COMPUTE VARIANCE/D12.5 = ((NUM*ASQ.X) - (X*X/NUM))/(NUM-1); NOPRINT
  COMPUTE STDEV/D12.5 = SQRT(VARIANCE); NOPRINT
  PRINT SIZE X NOPRINT
  -* COMPUTE NORMALIZED VALUES AND USE AS INPUT TO NORMSDST FUNCTION
  -* THEN USE RETURNED VALUES AS INPUT TO NORMSINV FUNCTION
  -* AND CONVERT BACK TO DATA VALUES
  COMPUTE Z/D12.5 = (X - MEAN)/STDEV;
  COMPUTE NORMSD/D12.5 = NORMSDST(Z, 'D8');
  COMPUTE NORMSI/D12.5 = NORMSINV(NORMSD, 'D8');
  COMPUTE DSIZE/D12 = NORMSI * STDEV + MEAN;
  BY PRODUCT_ID NOPRINT
END
```

The output shows that NORMSINV is the inverse of NORMSDST and returns the original values:

<table>
<thead>
<tr>
<th>Size</th>
<th>Z</th>
<th>NORMSD</th>
<th>NORMSI</th>
<th>DSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>-0.07298</td>
<td>0.47091</td>
<td>-0.07298</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>-0.80273</td>
<td>0.21106</td>
<td>-0.80273</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>0.65678</td>
<td>0.74434</td>
<td>0.65678</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>0.91721</td>
<td>1.38654</td>
<td>24</td>
</tr>
<tr>
<td>20</td>
<td>0.65678</td>
<td>0.74434</td>
<td>0.65678</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>1.38654</td>
<td>0.91721</td>
<td>1.38654</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>-0.07298</td>
<td>0.47091</td>
<td>-0.07298</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>-0.80273</td>
<td>0.21106</td>
<td>-0.80273</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>-1.53249</td>
<td>0.06270</td>
<td>-1.53249</td>
<td>8</td>
</tr>
</tbody>
</table>

Using Functions 369
How to: Generate Reproducible Random Numbers

The PRDNOR and PRDUNI functions generate reproducible random numbers:

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

  If PRDNOR generates a large set of numbers, they have the following properties:

  - The numbers lie roughly on a bell curve, as shown in the following figure. The bell curve is highest at the 0 mark, meaning that there are more numbers closer to 0 than farther away.

  ![Bell Curve Diagram](image)

  - The average of the numbers is close to 0.
  - The numbers can be any size, but most are between 3 and -3.

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

In z/OS, the numbers do not reproduce.
**Syntax:**  
How to Generate Reproducible Random Numbers

\{PRDNOR | PRDUNI\}(seed, output)

where:

PRDNOR
Generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

PRDUNI
Generates reproducible double-precision random numbers uniformly distributed between 0 and 1.

**seed**
Numeric
Is the seed or the field that contains the seed, up to 9 digits. The seed is truncated to an integer.

On z/OS, the numbers do not reproduce.

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

**Example:**  
Generating Reproducible Random Numbers

PRDNOR assigns random numbers and stores them in RAND. These values are then used to randomly pick five employee records identified by the values in the LAST NAME and FIRST NAME fields. The seed is 40. To produce a different set of numbers, change the seed.

```
DEFINE FILE EMPLOYEE
RAND/D12.2 WITH LAST_NAME = PRDNOR(40, RAND);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME
BY HIGHEST 5 RAND
END
```

The output is:

<table>
<thead>
<tr>
<th>RAND</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.38</td>
<td>STEVENS</td>
<td>ALFRED</td>
</tr>
<tr>
<td>1.12</td>
<td>MCCOY</td>
<td>JOHN</td>
</tr>
<tr>
<td>.55</td>
<td>SMITH</td>
<td>RICHARD</td>
</tr>
<tr>
<td>.21</td>
<td>JONES</td>
<td>DIANE</td>
</tr>
<tr>
<td>.01</td>
<td>IRVING</td>
<td>JOAN</td>
</tr>
</tbody>
</table>
RDNORM and RDUNIF: Generating Random Numbers

How to:
Generate Random Numbers

The RDNORM and RDUNIF functions generate random numbers:

- RDNORM generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
  
  If RDNORM generates a large set of numbers (between 1 and 32768), they have the following properties:
  - The numbers lie roughly on a bell curve, as shown in the following figure. The bell curve is highest at the 0 mark, meaning that there are more numbers closer to 0 than farther away.

  ![Frequency of Occurrence](image)

  - The average of the numbers is close to 0.
  - The numbers can be any size, but most are between 3 and -3.

- RDUNIF generates double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).
**Syntax:** How to Generate Random Numbers

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

where:

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

**RDUNIF**
Generates double-precision random numbers uniformly distributed between 0 and 1.

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

**Example:** Generating Random Numbers

RDNORM assigns random numbers and stores them in RAND. These numbers are then used to randomly choose five employee records identified by the values in the LAST NAME and FIRST NAME fields.

```verbatim
DEFINE FILE EMPLOYEE
RAND/D12.2 WITH LAST_NAME = RDNORM RAND;
END
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND FIRST_NAME
BY HIGHEST 5 RAND
END
```

The request produces output similar to the following:

<table>
<thead>
<tr>
<th>RAND</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>.65</td>
<td>CROSS</td>
<td>BARBARA</td>
</tr>
<tr>
<td>.20</td>
<td>BANNING</td>
<td>JOHN</td>
</tr>
<tr>
<td>.19</td>
<td>IRVING</td>
<td>JOAN</td>
</tr>
<tr>
<td>.00</td>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
</tr>
<tr>
<td>-.14</td>
<td>GREENSPAN</td>
<td>MARY</td>
</tr>
</tbody>
</table>

**SQRT: Calculating the Square Root**

**How to:**
Calculate 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}

Numeric

Is the value for which the square root is calculated, the name of a field that contains the value, or an expression that returns the value. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If you supply a negative number, the result is zero.

Example: Calculating the Square Root

SQRT calculates the square root of LISTPR:

\[
\text{TABLE FILE MOVIES}
\]

\[
\text{PRINT LISTPR AND COMPUTE SQRT\_LISTPR/D12.2 = SQRT(LISTPR)};
\]

\[
\text{BY TITLE}
\]

\[
\text{WHERE CATEGORY EQ 'MUSICALS'};
\]

\[
\text{END}
\]

The output is:

<table>
<thead>
<tr>
<th>TITLE</th>
<th>LISTPR</th>
<th>SQRT_LISTPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL THAT JAZZ</td>
<td>19.98</td>
<td>4.47</td>
</tr>
<tr>
<td>CABARET</td>
<td>19.98</td>
<td>4.47</td>
</tr>
<tr>
<td>CHORUS LINE, A</td>
<td>14.98</td>
<td>3.87</td>
</tr>
<tr>
<td>FIDDLER ON THE ROOF</td>
<td>29.95</td>
<td>5.47</td>
</tr>
</tbody>
</table>

XIRR: Calculating the Modified Internal Return Rate (Periodic or Non-Periodic)

How to:

Calculate the Internal Rate of Return

Reference:

Usage Notes for the XIRR Function

The XIRR function calculates the internal rate of return for a series of cash flows that can be periodic or non-periodic.
Syntax: How to Calculate the Internal Rate of Return

```
TABLE FILE ...
{PRINT|SUM} field ...
COMPUTE rrate/fmt = XIRR (cashflow, dates,guess, maxiterations, output);
WITHIN {sort_field|TABLE}
```

where:

- `field` ...
  Are fields that appear in the report output.

- `rrate`
  Is the field that contains the calculated return rate.

- `fmt`
  Is the format of the return rate. The data type must be D.

- `cashflow`
  Is a numeric field. Each value of this field represents either a payment (negative value) or income (positive value) for one period. The values must be in the correct sequence in order for the sequence of cash flows to be calculated correctly. The dates corresponding to each cash flow should be equally spaced and sorted in chronological order. The calculation requires at least one negative value and one positive value in the `cashflow` field. If the values are all positive or all negative, a zero result is returned.

- `dates`
  Is a date field containing the cash flow dates. The dates must be full component dates with year, month, and day components. Dates cannot be stored in fields with format A, I, or P. They must be stored in date fields (for example, format YMD, not AYMD). There must be the same number of dates as there are cash flow values. The number of dates must be the same as the number of cash flows.

- `guess`
  Is an (optional) initial estimate of the expected return rate expressed as a decimal. The default value is .1 (10%). To accept the default, supply the value 0 (zero) for this argument.

- `maxiterations`
  Is an (optional) number specifying the maximum number of iterations that can be used to resolve the rate using Newton's method. 50 is the default value. To accept the default, supply the value 0 (zero) for this argument. The rate is considered to be resolved when successive iterations do not differ by more than 0.0000003. If this level of accuracy is achieved within the maximum number of iterations, calculation stops at that point. If it is not achieved after reaching the maximum number of iterations, calculation stops and the value calculated by the last iteration is returned.
output

D

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

sort_field

Is a field that sorts the report output and groups it into subsets of rows on which the function can be calculated separately. To calculate the function using every row of the report output, use the WITHIN TABLE phrase. A WITHIN phrase is required.

Reference: Usage Notes for the XIRR Function

- This function is only supported in a COMPUTE command with the WITHIN phrase.
- The cash flow field must contain at least one negative value and one positive value.
- Dates cannot be stored in fields with format A, I, or P. They must be stored in date fields (for example, format YMD, not AYMD).
- Cash flows or dates with missing values are not supported.

Example: Calculating the Internal Rate of Return

The following request creates a FOCUS data source with cash flows and dates and calculates the internal return rate.

The Master File for the data source is:

```sql
FILENAME=XIRR01,SUFFIX=FOC
SEGNAME=SEG1,SEGTYPE=S1
FIELDNAME=DUMMY,FORMAT=A2,$
FIELDNAME=DATES,FORMAT=YYMD,$
FIELDNAME=CASHFL,FORMAT=D12.4,$
END
```

The procedure to create the data source is:

```sql
CREATE FILE XIRR01
MODIFY FILE XIRR01
FREEFORM DUMMY DATES CASHFL
DATA
AA,19980101,-10000. ,$
BB,19980301,2750.   ,$
CC,19981030,4250.   ,$
DD,19990215,3250.   ,$
EE,19990401,2750.   ,$
END
```
The request is sorted by date so that the correct cash flows can be calculated. The rate returned by the function is multiplied by 100 in order to express it as a percent rather than a decimal value. Note that the format includes the % character. This causes a percent symbol to display, but it does not calculate a percent:

```
TABLE FILE XIRR01
PRINT CASHFL
COMPUTE RATEX/D12.2%=XIRR(CASHFL, DATES, 0., 0., RATEX) * 100;
WITHIN TABLE
BY DATES
END
```

One rate is calculated for the entire report because of the WITHIN TABLE phrase:

<table>
<thead>
<tr>
<th>DATES</th>
<th>CASHFL</th>
<th>RATEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998/01/01</td>
<td>-10,000.00</td>
<td>37.49%</td>
</tr>
<tr>
<td>1998/03/01</td>
<td>2,750.00</td>
<td>37.49%</td>
</tr>
<tr>
<td>1998/10/30</td>
<td>4,250.00</td>
<td>37.49%</td>
</tr>
<tr>
<td>1999/02/15</td>
<td>3,250.00</td>
<td>37.49%</td>
</tr>
<tr>
<td>1999/04/01</td>
<td>2,750.00</td>
<td>37.49%</td>
</tr>
</tbody>
</table>
System functions call the operating system to obtain information about the operating environment or to use a system service.

For many functions, the output argument can be supplied either as a field name or as a format enclosed in single quotation marks. However, if a function is called from a Dialogue Manager command, this argument must always be supplied as a format, and if a function is called from a Maintain procedure, this argument must always be supplied as a field name. For detailed information about calling a function and supplying arguments, see Accessing and Calling a Function on page 43.

Topics:
- CLSDDREC: Closing All Files Opened by the PUTDDREC Function
- FEXERR: Retrieving an Error Message
- FINDMEM: Finding a Member of a Partitioned Data Set
- GETPDS: Determining If a Member of a Partitioned Data Set Exists
- GETUSER: Retrieving a User ID
- MVSDYNAM: Passing a DYNAM Command to the Command Processor
- 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
CLSDREC: Closing All Files Opened by the PUTDDREC Function

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

The CLSDREC 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 files by calling the CLSDREC function. For information about PUTDDREC, see PUTDDREC: Writing a Character String as a Record in a Sequential File on page 391.

**Syntax:**

```
CLSDREC(output)
```

where:

- `output`
  
  Integer
  
  Is the return code, which can be one of the following values:
  - 0 - Files are closed.
  - 1 - Error while closing the files.

**Example:**

Closing Files Opened by the PUTDDREC Function

This example closes files opened by the PUTDDREC function:

```
CLSDREC('I1')
```

FEXERR: Retrieving an Error Message

**How to:**

Retrieve an Error Message

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

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

FEXERR(error, 'A72')

where:

error

Numeric

Is the error number, up to 5 digits long.

'A72'

Is the format of the output value enclosed in single quotation marks. The format is A72, the maximum length of an Information Builders error message.

Example: Retrieving an Error Message

FEXERR retrieves the error message whose number is contained in the &ERR variable, in this case 650. The result is returned to the variable &&MSGVAR and has the format A72.

-SET &ERR = 650;
-SET &&MSGVAR = FEXERR(&ERR, 'A72');
-TYPE &&MSGVAR

The output is:

(FOC650) THE DISK IS NOT ACCESSED

FINDMEM: Finding a Member of a Partitioned Data Set

How to:
Find a Member of a Partitioned Data Set

Available Operating Systems: z/OS

The FINDMEM function, available only on z/OS, determines if a specific member of a partitioned data set (PDS) exists. This function is used primarily in Dialogue Manager procedures.

To use this function, allocate the PDS to a ddname because the ddname is required in the function call. You can search multiple PDSs with one function call if they are concatenated to one ddname.
**Syntax:**

FINDMEM(ddname, member, output)

where:

- **ddname**
  - A8
  - Is the ddname to which the PDS is allocated. This value must be an eight-character literal enclosed in single quotation marks, or a variable that contains the ddname. If you supply a literal less than eight characters long, pad it with trailing spaces.

- **member**
  - A8
  - Is the member for which you are searching. This value must be eight characters long. If you supply a literal that has less than eight characters, pad it with trailing spaces.

- **output**
  - A1
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The result is one of the following:

    - **Y** indicates the member exists in the PDS.
    - **N** indicates the member does not exist in the PDS.
    - **E** indicates an error occurred. Either the data set is not allocated to the ddname, or the data set allocated to the ddname is not a PDS (and may be a sequential file).
Example: **Finding a Member of a Partitioned Data Set**

FINDMEM searches for the EMPLOYEE Master File in the PDS allocated to ddname MASTER, and returns the result to the variable &FINDCODE. The result has the format A1:

```
-SET &FINDCODE = FINDMEM('MASTER ', 'EMPLOYEE', 'A1'); -IF &FINDCODE EQ 'N' GOTO NOMEM;
-IF &FINDCODE EQ 'E' GOTO NOPDS;
-TYPE MEMBER EXISTS, RETURN CODE = &FINDCODE
 TABLE FILE EMPLOYEE
PRINT CURR_SAL BY LAST_NAME BY FIRST_NAME
WHERE RECORDLIMIT EQ 4;
END
-EXIT
-NOMEM
-TYPE EMPLOYEE NOT FOUND IN MASTER FILE PDS
-EXIT
-NOPDS
-TYPE ERROR OCCURRED IN SEARCH
-TYPE CHECK IF FILE IS A PDS ALLOCATED TO DDNAME MASTER
-EXIT
```

The output is:

```
MEMBER EXISTS, RETURN CODE = Y
> NUMBER OF RECORDS IN TABLE= 4 LINES= 4
LAST_NAME    FIRST_NAME   CURR_SAL
-------    ----------    --------
JONES      DIANE      $18,480.00
SMITH      MARY      $13,200.00
           RICHARD    $9,500.00
STEVENS    ALFRED    $11,000.00
```

**GETPDS: Determining If a Member of a Partitioned Data Set Exists**

**How to:**

Determine If a PDS Member Exists

Available Operating Systems: z/OS

The GETPDS function determines if a specific member of a partitioned data set (PDS) exists, and if it does, returns the PDS name. This function is used primarily in Dialogue Manager procedures.

To use this function, allocate the PDS to a ddname because the ddname is required in the function call. You can search multiple PDSs with one function call if they are concatenated to one ddname.
GETPDS is almost identical to FINDMEM, except that GETPDS provides either the PDS name or returns a different set of status codes.

**Syntax:**  
**How to Determine If a PDS Member Exists**

GETPDS (ddname, member, output)

where:

- **ddname**
  - A8
  - Is the ddname to which the PDS is allocated. This value must be an eight-character literal enclosed in single quotation marks, or a variable that contains the ddname. If you supply a literal less than eight characters long, pad it with trailing spaces.

- **member**
  - A8
  - Is the member for which the function searches. This value must be eight characters long. If you supply a literal with less than eight characters, pad it with trailing spaces.

- **output**
  - A44
  - Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks. The maximum length of a PDS name is 44. The result is one of the following:

  - **PDS name** is the name of the PDS that contains the member, if it exists.
  - **D** indicates the ddname is not allocated to a data set.
  - **M** indicates the member does not exist in the PDS.
  - **E** indicates an error occurred. For example, the data set allocated to the ddname is not a PDS (and may be a sequential file).
Example: Determining If a PDS Member Exists

GETPDS searches for the member specified by &MEMBER in the PDS allocated to &DDNAME, and returns the result to &PNAME. The result has the format A44.

-SET &DDNAME = 'MASTER ';
-SET &MEMBER = 'EMPLOYEE';
-SET &PNAME = '                                            ';
-SET &PNAME = GETPDS(&DDNAME, &MEMBER, 'A44');
-IF &PNAME EQ '*D' THEN GOTO DDNOAL;
-IF &PNAME EQ '*M' THEN GOTO MEMNOF;
-IF &PNAME EQ '*E' THEN GOTO DDERROR;
-*/
-TYPE MEMBER &MEMBER IS FOUND IN
-TYPE THE PDS &PNAME
-TYPE ALLOCATED TO &DDNAME
-*/
-EXIT
-DDNOAL
-*/
-TYPE DDNAME &DDNAME NOT ALLOCATED
-*/
-EXIT
-MEMNOF
-*/
-TYPE MEMBER &MEMBER NOT FOUND UNDER DDNAME &DDNAME
-*/
-EXIT
-DDERROR
-*/
-TYPE ERROR IN GETPDS; DATA SET PROBABLY NOT A PDS.
-*/
-EXIT

The output is similar to the following:

MEMBER EMPLOYEE IS FOUND IN
THE PDS USER1.MASTER.DATA
ALLOCATED TO MASTER
**Example: Copying a Member for Editing in TED**

GETPDS searches for the member specified by &MEMBER in the PDS allocated to &DDNAME, and returns the result to &PNAME. The DYNAM commands copy the member from the production PDS to the local PDS. Then the TED editor enables you to edit the member. The ddnames are allocated earlier in the session: the production PDS is allocated to the ddbname MASTER; the local PDS to ddbname MYMASTER.

```plaintext
-SET &DDNAME = 'MASTER  
-SET &MEMBER = &MEMBER; 
-SET &PNAME = ' 
-SET &PNAME = GETPDS(&DDNAME, &MEMBER, 'A44'); IF &PNAME EQ '*D' OR '*M' OR '*E' THEN GOTO DDERROR; 
* DYNAM ALLOC FILE XXXX DA - 
   &PNAME MEMBER &MEMBER SHR 
DYNAM COPY XXXX MYMASTER MEMBER &MEMBER 
RUN 
TED MYMASTER(&MEMBER) 
EXIT 
* DDERROR 
* -TYPE Error in GETPDS; Check allocation for &DDNAME for 
-TYPE proper allocation. 
* EXIT 
```

Earlier in the session, allocate the ddnames:

```plaintext
> > tso alloc f(master) da('prod720.master.data') shr 
> > tso alloc f(mymaster) da('user1.master.data') shr 
```
Run the procedure, and specify the EMPLOYEE member. It is copied to your local PDS, and you access TED.

PLEASE SUPPLY VALUES REQUESTED

MEMBER = EMPLOYEE

Example: Displaying the Attributes of a PDS

To view the attributes of the PDS that contains a specific member, this Dialogue Manager procedure can search for the EMPLOYEE member in the PDS allocated to the ddname MASTER and, based on its existence, allocate the PDS to the ddname TEMPMAST. Dialogue Manager system variables are used to display the attributes.

-SET &DDNAME = 'MASTER ';
-SET &MEMBER = 'EMPLOYEE';
-SET &PNAME = '';
-SET &PNAME = GETPDS(&DDNAME, &MEMBER, 'A44');
-IF &PNAME EQ '*D' OR '*M' OR '*E' THEN GOTO DDERROR;
- *
DYNAM ALLOC FILE TEMPMAST DA -
   &PNAME SHR
- RUN
- ? MVS DDNAME TEMPMAST
- TYPE The data set attributes include:
- TYPE Data set name is: &DSNAME
- TYPE Volume is: &VOLSER
- TYPE Disposition is: &DISP
- EXIT
- *
- DDERROR
- TYPE Error in GETPDS; Check allocation for &DDNAME for
- TYPE proper allocation.
- *
- EXIT
The sample output is:

THE DATA SET ATTRIBUTES INCLUDE:
DATA SET NAME IS: USER1.MASTER.DATA
VOLUME IS: USERM0
DISPOSITION IS: SHR

GETUSER: Retrieving a User ID

How to:
Retrieve a User ID

The GETUSER function retrieves the ID of the connected user. GETUSER can also retrieve the name of a z/OS batch job if you run the function from the batch job. To retrieve a logon ID for MSO, use the MSOINFO function described in the *FOCUS for IBM Mainframe Multi-Session Option Installation and Technical Reference Guide*.

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 request:

```sql
DEFINE FILE EMPLOYEE
USERID/A8 WITH EMP_ID = GETUSER(USERID);
END

TABLE FILE EMPLOYEE
SUM CURR_SAL AS 'TOTAL SALARIES' BY DEPARTMENT
HEADING
"SALARY REPORT RUN FROM USERID: <USERID"
""
END
```
The output is:

```
SALARY REPORT RUN FROM USERID: USER1

DEPARTMENT    TOTAL SALARIES
--------------    --------------
MIS             $108,002.00
PRODUCTION     $114,282.00
```

**MVSDYNAM: Passing a DYNAM Command to the Command Processor**

**How to:**

Pass a DYNAM Command to the Command Processor

Available Operating Systems: z/OS

The MVSDYNAM function transfers a FOCUS DYNAM command to the DYNAM command processor. It is useful in passing allocation commands to the processor in a compiled MODIFY procedure after the CASE AT START command.

**Syntax:**

```
MVSDYNAM(command, length, outfield)
```

where:

- **command**
  - Alphanumeric
  - Is the DYNAM command enclosed in single quotation marks, or a field or variable that contains the command. The function converts lowercase input to uppercase.

- **length**
  - Numeric
  - Is the maximum length of the command, in characters, between 1 and 256.

- **outfield**
  - I4
  - Is the field that contains the result, or the format of the output value enclosed in single quotation marks.

MVSDYNAM returns one of the following codes:

- **0** indicates the DYNAM command transferred and executed successfully.
- **positive number** is the error number corresponding to a FOCUS error.
- **negative number** is the FOCUS error number corresponding to a DYNAM failure.
In Dialogue Manager, you must specify the format.

**Example:**  **Passing a DYNAM Command to the Command Processor**

MVSDYNAM passes the DYNAM command contained in LINE to the processor. The return code is stored in RES.

```plaintext
-* THE RESULT OF ? TSO DDNAME CAR WILL BE BLANK AFTER ENTERING
-* 'FREE FILE CAR' AS YOUR COMMAND
DYNAM ALLOC FILE CAR DS USER1.CAR.FOCUS SHR REUSE
? TSO DDNAME CAR
-RUN
-PROMPT &XX.ENTER A SPACE TO CONTINUE.
MODIFY FILE CAR
COMPUTE LINE/A60=;
    RES/I4 = 0;
CRTFORM
" ENTER DYNAM COMMAND BELOW:"
" <LINE>"
COMPUTE
RES = MVSDYNAM(LINE, 60, RES);GOTO DISPLAY
    CASE DISPLAY
    CRTFORM LINE 1
" THE RESULT OF DYNAM WAS <D.RES"
GOTO EXIT
ENDCASE
DATA
END
? TSO DDNAME CAR
```

The first query command displays the allocation that results from the DYNAM ALLOC command:

```plaintext
DDNAME    = CAR
DSNAME    = USER1.CAR.FOCUS
DISP      = SHR
DEVICE    = DISK
VOLSER    = USERMN
DSORG     = PS
RECFM     = F
SECONDARY = 100
ALLOCATION = BLOCKS
BLKSIZE   = 4096
LRECL     = 4096
TRKTOT    = 8
EXTENTSUSED = 1
BLKSPERTRK = 12
TRKSPERCYL = 15
CYLSPERDISK = 2227
BLKSWRITTEN = 96
FOCUSPAGES = 8
ENTER A SPACE TO CONTINUE >
```
Type one space and press Enter to continue. Then enter the DYNAM FREE command (the DYNAM keyword is assumed):

```
ENTER DYNAM COMMAND BELOW:
  free file car
```

The function successfully passes the DYNAM FREE command to the processor and the return code displays:

```
THE RESULT OF DYNAM WAS     0
```

Press Enter to continue. The second query command indicates that the allocation was freed:

```
DDNAME      =  CAR
DSNAME      =
DISP        =
DEVICE      =
VOLSER      =
DSORG       =
RECFM       =
SECONDARY   =  ****
ALLOCATION  =
BLKSIZE     =         0
LRECL       =         0
TRKTOT      =         0
EXTENTSUSED =         0
BLKSPERTRK  =         0
TRKSPERCYL  =         0
CYLSPERDISK =         0
BLKSWRITTEN =         0
> 
```

**PUTDDREC: Writing a Character String as a Record in a Sequential File**

**How to:**

**Write a Character String as a Record in a Sequential File**

The PUTDDREC function writes a character string as a record in a sequential file. The file must be identified with a DYNAM command. TSO ALLOCATE does not work. If the file is defined as an existing file (with the APPEND option), the new record is appended. If the file is defined as NEW and it already exists, the new record overwrites the existing file.

For information about the DYNAM command, see the *Overview and Operating Environments* manual.
PUTDDREC opens the file if it is not already open. Each call to PUTDDREC can use the same file or a new one. All of the files opened by PUTDDREC remain open until the end of a request or session. At the end of the request or session, all files opened by PUTDDREC are automatically closed. For information about closing files opened by PUTDDREC in order to free the memory used, see CLSDREC: Closing All Files Opened by the PUTDDREC Function on page 380.

- The open, close, and write operations are handled by the operating system. Therefore, the requirements for writing to the file and the results of deviating from the instructions when calling PUTDDREC are specific to your operating environment. Make sure you are familiar with and follow the guidelines for your operating system when performing input/output operations.

- You can call PUTDDREC in a DEFINE FILE command or in a DEFINE in the Master File. However, PUTDDREC does not open the file until its field name is referenced in a request.

If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or session. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDREC 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 DYNAM 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 DYNAM 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: Calling PUTDDREC in a TABLE Request**

The following example defines a new file whose logical name is PUTDD1. The TABLE request then calls PUTDDREC for each employee in the EMPLOYEE data source and writes a record to the file composed of the employee's last name, first name, employee ID, current job code, and current salary (converted to alphanumeric using the EDIT function). The return code of zero (in OUT1) indicates that the calls to PUTDDREC were successful:

```
DYNAM ALLOC PUTDD1 DA USER1.PUTDD1.DATATABLE FILE EMPLOYEE
PRINT EMP_ID CURR_JOBCODE AS 'JOB' CURR_SAL
COMPUTE SALA/A12 = EDIT(CURR_SAL); NOPRINT
COMPUTE EMP1/A50= LAST_NAME|FIRST_NAME|EMP_ID|CURR_JOBCODE|SALA;
NPRINT
COMPUTE OUT1/I1 = PUTDDREC('PUTDD1',6, EMP1, 50, OUT1);
BY LAST_NAME BY FIRST_NAME
END
```

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMP_ID</th>
<th>JOB</th>
<th>CURR_SAL</th>
<th>OUT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>119329144</td>
<td>A17</td>
<td>$29,700.00</td>
<td>0</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357</td>
<td>B04</td>
<td>$21,780.00</td>
<td>0</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173</td>
<td>A17</td>
<td>$27,062.00</td>
<td>0</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165</td>
<td>A07</td>
<td>$9,000.00</td>
<td>0</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>123764317</td>
<td>A15</td>
<td>$26,862.00</td>
<td>0</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129</td>
<td>B03</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371</td>
<td>B02</td>
<td>$18,480.00</td>
<td>0</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>451123478</td>
<td>B02</td>
<td>$16,100.00</td>
<td>0</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>126724188</td>
<td>B04</td>
<td>$21,120.00</td>
<td>0</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>119265415</td>
<td>A01</td>
<td>$9,500.00</td>
<td>0</td>
</tr>
<tr>
<td>STEEVENS</td>
<td>ALFRED</td>
<td>071382660</td>
<td>A07</td>
<td>$11,000.00</td>
<td>0</td>
</tr>
</tbody>
</table>
After running this request, the sequential file contains the following records:

<table>
<thead>
<tr>
<th>Name</th>
<th>Last Name</th>
<th>SSN</th>
<th>Age</th>
<th>Record Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>119329144A17000000029700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>326179357B0400000021780</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>818692173A1700000027062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>543729165A0700000009000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>123764317A1500000026862</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>117593129B0300000018480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>219984371B0200000018480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>451123478B0200000016100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>126724188B0400000021120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>112847612B1400000013200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>119265415A0100000009500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>071382660A0700000011000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Calling PUTDDREC and CLSDDREC in Dialogue Manager -SET Commands

The following example defines a new file whose logical name is PUTDD1. The first -SET command creates a record to add to this file. The second -SET command calls PUTDDREC to add the record. The last -SET command calls CLSDDREC to close the file. The return codes are displayed to make sure operations were successful:

```plaintext
DYNAM ALLOC PUTDD1 DA USER1.PUTDD1.DATA -SET &EMP1 = 'SMITH'|'MARY'|'A07'|'27000';
-TYPE DATA = &EMP1
-SET &OUT1 = PUTDDREC('PUTDD1',6, &EMP1, 17, 'I1');
-TYPE PUT RESULT = &OUT1
-SET &OUT1 = CLSDDREC('I1');
-TYPE CLOSE RESULT = &OUT1
```

The output is:

```
DATA = SMITHMARYA0727000
PUT RESULT = 0
CLOSE RESULT = 0
```

After running this procedure, the sequential file contains the following record:

```
SMITHMARYA0727000
```

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

**How to:**

Suspend Execution for a Specified Number of Seconds

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

This function is most useful in Dialogue Manager when you need to wait to start a specific procedure. For example, you can start a FOCUS Database Server and wait until the server is started before initiating a client application.
**Syntax:**  How to Suspend Execution for a Specified Number of Seconds

SLEEP(*delay*, *output*);

where:

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

*output*  
Numeric  
Is the name of a field or a format enclosed in single quotation marks. The value returned is the same value you specify for delay.

**Example:**  Suspending Execution for Four Seconds

The following example computes the current date and time, suspends execution for 4 seconds, and computes the current date and time after the delay:

```
TABLE  FILE VIDEOTRK
PRINT TRANSDATE NOPRINT
COMPUTE
   START_TIME/HYYMDSa = HGETC(8, START_TIME);
   DELAY/I2 = SLEEP(4.0, 'I2');
   END_TIME/HYYMDSa = HGETC(8, END_TIME);
   IF RECORDLIMIT EQ 1
END
```

The output is:

```
START_TIME     DELAY     END_TIME
----------     -----     --------
2007/10/26  5:04:36pm  4  2007/10/26  5:04:40pm
```
SYSVAR: Retrieving the Value of a z/OS System Variable

How to:
Retrieve the Value of a z/OS System Variable

Available Operating Systems: z/OS

The SYSVAR function populates a Dialogue Manager amper variable with the contents of any z/OS system variable. System variables are in the format [&]name[.], where the dot is optional. They can be provided by the operating system or can be user defined. The function can be called in a -SET command.

Syntax: How to Retrieve the Value of a z/OS System Variable

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

where:

&dmvar
Alphanumeric
Is the name of the Dialogue Manager variable to be populated with the value of the z/OS system variable.

length
Alphanumeric
Is the length of the next parameter in the call. Do not include the escape character in the length, if one is present in the sysvar argument.

[&]sysvar[.]
Alphanumeric
Is the name of the system variable to be retrieved. Note that the ampersand (&) and the dot (.) are optional. If the ampersand is included, it must be followed by the escape character (|).

outfmt
Alphanumeric
Is the format of the returned value enclosed in single quotation marks.
Example: Retrieving the Value of the z/OS SYSNAME Variable

The following example populates the Dialogue Manager variable named &MYSNAME2 with the value of the z/OS SYSNAME variable:

-SET &MYSNAME2=SYSVAR('7','SYSNAME','A8');
- TYPE SYSNAME:&MYSNAME2

The output is similar to the following:

SYSNAME:IBI1
Creating a Subroutine

You can create custom subroutines to use in addition to the functions provided by Information Builders. The process of creating a subroutine consists of the following steps:

- **Writing a Subroutine** using any language that supports subroutine calls. Some of the most common languages are FORTRAN, COBOL, PL/I, Assembler, and C. For details, see *Writing a Subroutine* on page 400.

- **Compiling the subroutine**. For details, see *Compiling and Storing a Subroutine* on page 411.

- **Storing the subroutine** in a separate file; do not include it in the main program. For details, see *Compiling and Storing a Subroutine* on page 411.

- **Testing the subroutine**. For details, see *Testing the Subroutine* on page 411.

**Note:** On z/OS, all subroutines called by FOCUS must be fully LE compliant.

---

**Topics:**

- Writing a Subroutine
- Compiling and Storing a Subroutine
- Testing the Subroutine
- Using a Custom Subroutine: The MTHNAM Subroutine
- Subroutines Written in REXX
You can write a subroutine in any language that supports subroutines. If you intend to make your subroutine available to other users, be sure to document what your subroutine does, what the arguments are, what formats they have, and in what order they must appear in the subroutine call.

When you write a subroutine you need to consider the requirements and limits that affect it. These are:

- Naming conventions. For details, see Naming a Subroutine on page 401.
- Argument considerations. For details, see Creating Arguments on page 401.
- Language considerations. For details, see Language Considerations on page 402.
- Programming considerations. For details, see Programming a Subroutine on page 405.

If you write a program named INTCOMP that calculates the amount of money in an account earning simple interest, the program reads a record, tests if the data is acceptable, and then calls a subroutine called SIMPLE that computes the amount of money. The program and the subroutine are stored together in the same file.

The program and the subroutine shown here are written in pseudocode (a method of representing computer code in a general way):

```plaintext
Begin program INTCOMP.
Execute this loop until end-of-file.
    Read next record, fields: PRINCIPAL, DATE_PUT, YRRATE.
    If PRINCIPAL is negative or greater than 100,000, reject record.
    If DATE_PUT is before January 1, 1975, reject record.
    If YRRATE is negative or greater than 20%, reject record.
    Call subroutine SIMPLE (PRINCIPAL, DATE_PUT, YRRATE, TOTAL).
    Print PRINCIPAL, YEARRATE, TOTAL.
End of loop.
End of program.
```
Subroutine SIMPLE (AMOUNT, DATE, RATE, RESULT).
Retrieve today's date from the system.
Let NO_DAYS = Days from DATE until today's date.
Let DAY_RATE = RATE / 365 days in a year.
Let RESULT = AMOUNT * (NO_DAYS * DAY_RATE + 1).
End of subroutine.

If you move the SIMPLE subroutine into a file separate from the main program and compile it, you can call the subroutine. The following report request shows how much money employees would accrue if they invested salaries in accounts paying 12%:

TABLE FILE EMPLOYEE
PRINT LAST_NAME DAT_INC SALARY AND COMPUTE
  INVESTED/D10.2 = SIMPLE(SALARY, DAT_INC, 0.12, INVESTED);
BY EMP_ID
END

Note: The subroutine is designed to return only the amount of the investment, not the current date because a subroutine can return only a single value each time it is called.

Naming a Subroutine

A subroutine name can be up to eight characters long unless the language you are using to write the subroutine requires a shorter name. A name must start with a letter and can consist of a combination of letters and/or numbers. Special symbols are not permitted.

Creating Arguments

When you create arguments for a subroutine, you must consider the following issues:

- **Maximum number of arguments.** A subroutine may contain up to 200 arguments. You can bypass this restriction by creating a subroutine that accepts multiple calls as described in *Including More Than 200 Arguments in a Subroutine Call* on page 407.

- **Argument types.** You can use the same types of arguments in a subroutine as in a function. For details on these argument types, see *Argument Types* on page 46.

- **Input arguments.** Input arguments are passed to a subroutine using standard conventions. Register one point to the list of arguments.

  The input parameter values should not be changed in the subroutine code. You should not assume that the input parameters are stored in contiguous memory. The same parameter cannot be used for both input and output.

- **Output arguments.** A subroutine returns only one output argument. This argument must be the last in the subroutine. You can choose any format for the output argument except in Dialogue Manager which requires the argument to have the format of the output field.
Internal processing. A subroutine’s arguments are processed as follows:

- An alphanumeric argument is not changed.
- A numeric argument is converted to floating-point double-precision format except in an operating system RUN command or when storing the output in a variable.

Dialogue Manager requirements. If you are writing a subroutine specifically for Dialogue Manager, the subroutine may need to perform a conversion. For details on using a subroutine with Dialogue Manager, see Calling a Function From a Dialogue Manager Command on page 53.

The lengths of the calling arguments as defined in FOCUS must match the lengths of the corresponding arguments defined in the subroutine.

Any deviation from these rules may result in problems in using the subroutine. Information Builders recommends that you modify the subroutine to conform to the stated rules and then link it above the line. In order to load subroutines above the line, the following are the required link-edit options for compiling and storing the subroutine:

- AMODE 31 (Addressing Mode - 31-bit addressing)
- RMODE ANY (System can load this routine anywhere)

Language Considerations

When writing a subroutine, you must consider the following language issues:

Language and memory. If you write a subroutine in a language that brings libraries into memory (for example, FORTRAN and COBOL), the libraries reduce the amount of memory available to the subroutine.

FORTRAN. TSO supports FORTRAN input/output operations.

COBOL. When writing a subroutine in COBOL:

- The subroutine must use the GOBACK command to return to the calling program. STOPRUN is not supported.
- Numeric arguments received from a request must be declared as COMP-2 (double precision floating point).
- The format described in the DEFINE or COMPUTE command determines the format of the output argument:

<table>
<thead>
<tr>
<th>FOCUS Format</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>Xn</td>
</tr>
</tbody>
</table>
**FOCUS Format** | **Picture**
---|---
I | S9(9) COMP
P | S9(n) [V9(m)]
   where:
   
   \[
   \frac{1+n+m}{2} = 8 \quad \text{for small packed numbers.}
   \]
   
   \[
   \frac{1+n+m}{2} = 16 \quad \text{for large packed numbers.}
   \]
D | COMP-2
F | COMP-1

**PL/I.** When writing a subroutine in PL/I:

- The RETURNS attribute cannot be used.
- The following attribute must be in the procedure (PROC) statement:
  
  \[\text{OPTIONS (COBOL)}\]
- Alphanumeric arguments received from a request must be declared as
  
  \[\text{CHARACTER (n)}\]
  
  where:
  
  \[n\]
  
  Is the field length as defined by the request. Do not use the VARYING attribute.
- Numeric arguments received from a request must be declared as
  
  \[\text{DECIMAL FLOAT (16)}\]
  
  or
  
  \[\text{BINARY FLOAT (53)}\]
The format described in the DEFINE or COMPUTE command determines the format of the output argument:

<table>
<thead>
<tr>
<th>FOCUS Format</th>
<th>PL/I Declaration for Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>CHARACTER (n)</td>
</tr>
<tr>
<td>I</td>
<td>BINARY FIXED (31)</td>
</tr>
<tr>
<td>F</td>
<td>DECIMAL FLOAT (6) or BINARY FLOAT (21)</td>
</tr>
<tr>
<td>D</td>
<td>DECIMAL FLOAT (16) or BINARY FLOAT (53)</td>
</tr>
<tr>
<td>P</td>
<td>DECIMAL FIXED (15) (for small packed numbers, 8 bytes)</td>
</tr>
<tr>
<td></td>
<td>DECIMAL FIXED (31) (for large packed numbers, 16 bytes)</td>
</tr>
</tbody>
</table>

Variables that are not arguments with the STATIC attribute must be declared. This avoids dynamically allocating these variables every time the subroutine is executed.

C language. When writing a subroutine in C:

- Do not return a value with the return statement.
- Declare double-precision fields as Double.
- The format defined in the DEFINE or COMPUTE command determines the format of the output argument:

<table>
<thead>
<tr>
<th>FOCUS Format</th>
<th>C Declaration for Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>char *xxx n</td>
</tr>
<tr>
<td></td>
<td>Alphanumeric fields are not terminated with a null byte and cannot be processed by many of the string manipulation subroutines in the run-time library.</td>
</tr>
<tr>
<td>I</td>
<td>long *xxx</td>
</tr>
<tr>
<td>F</td>
<td>float *xxx</td>
</tr>
</tbody>
</table>
### Programming a Subroutine

**In this section:**
- Executing a Subroutine at an Entry Point
- Including More Than 200 Arguments in a Subroutine Call

Consider the following when planning your programming requirements:

- Write the subroutine to include an argument that specifies the output field.

- If the subroutine initializes a variable, it must initialize it each time it is executed (serial reusability).

- Since a single request may execute a subroutine numerous times, code the subroutine as efficiently as possible.

- If you create your subroutine in a text file or text library, the subroutine must be 31-bit addressable.

- The last argument, which is normally used for returning the result of the subroutine, can also be used to provide input from the subroutine.

You can add flexibility to your subroutine by using a programming technique. A programming technique can be one of the following:

- Executing a subroutine at an entry point. An entry point enables you to use one algorithm to produce different results. For details, see *Executing a Subroutine at an Entry Point* on page 406.

- Creating a subroutine with multiple subroutine calls. Multiple calls enable the subroutine to process more than 200 arguments. For details, see *Including More Than 200 Arguments in a Subroutine Call* on page 407.

<table>
<thead>
<tr>
<th>FOCUS Format</th>
<th>C Declaration for Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>double *xxx</td>
</tr>
<tr>
<td>P</td>
<td>No equivalent in C.</td>
</tr>
</tbody>
</table>
**Executing a Subroutine at an Entry Point**

**How to:**
Execute a Subroutine at an Entry Point

A subroutine is usually executed starting from the first statement. However, a subroutine can be executed starting from any place in the code designated as an *entry point*. This enables a subroutine to use one basic algorithm to produce different results. For example, the DOWK subroutine calculates the day of the week on which a date falls. By specifying the subroutine name DOWK, you obtain a 3-letter abbreviation of the day. If you specify the entry name DOWKL, you obtain the full name. The calculation, however, is the same.

Each entry point has a name. To execute a subroutine at an entry point, specify the entry point name in the subroutine call instead of the subroutine name. How you designate an entry point depends on the language you are using.

**Syntax:**

How to Execute a Subroutine at an Entry Point

```
{subroutine|entrypoint}  (input1, input2,...outfield)
```

where:

- **subroutine**
  - Is the name of the subroutine.

- **entrypoint**
  - Is the name of the entry point to execute the subroutine at.

- **input1, input2,...**
  - Are the subroutine’s arguments.

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

In Dialogue Manager, you must specify the format. In Maintain, you must specify the name of the field.
**Example:** Executing a Subroutine at an Entry Point

The FTOC subroutine, written in pseudocode below, converts Fahrenheit temperature to Centigrade. The entry point FTOK (designated by the Entry command) sets a flag that causes 273 to be subtracted from the Centigrade temperature to find the Kelvin temperature. The subroutine is:

```
Subroutine FTOC (FAREN, CENTI).
Let FLAG = 0.
Go to label X.
Entry FTOK (FAREN, CENTI).
Let FLAG = 1.
Label X.
Let CENTI = (5/9) * (FAREN - 32).
If FLAG = 1 then CENTI = CENTI - 273.
Return.
End of subroutine.
```

The following is a shorter way to write the subroutine. Notice that the kelv output argument listed for the entry point is different from the centi output argument listed at the beginning of the subroutine:

```
Subroutine FTOC (FAREN, CENTI).
Entry FTOK (FAREN, KELV).
Let CENTI = (5/9) * (FAREN - 32).
KELV = CENTI - 273.
Return.
End of Subroutine.
```

To obtain the Centigrade temperature, specify the subroutine name FTOC in the subroutine call. The subroutine processes as:

```
CENTIGRADE/D6.2 = FTOC (TEMPERATURE, CENTIGRADE);
```

To obtain the Kelvin temperature, specify the entry name FTOK in the subroutine call. The subroutine processes as:

```
KELVIN/D6.2 = FTOK (TEMPERATURE, KELVIN);
```

**Including More Than 200 Arguments in a Subroutine Call**

**How to:**
Create a Subroutine With Multiple Call Statements

A subroutine can specify a maximum of 200 arguments including the output argument. To process more than 200 arguments, the subroutine must specify two or more call statements to pass the arguments to the subroutine.
Use the following technique for writing a subroutine with multiple calls:

1. Divide the subroutine into segments. Each segment receives the arguments passed by one corresponding subroutine call.

   The argument list in the beginning of your subroutine must represent the same number of arguments in the subroutine call, including a call number argument and an output argument.

   Each call contains the same number of arguments. This is because the argument list in each call must correspond to the argument list in the beginning of the subroutine. You may process some of the arguments as dummy arguments if you have an unequal number of arguments. For example, if you divide 32 arguments among six segments, each segment processes six arguments; the sixth segment processes two arguments and four dummy arguments.

   Subroutines may require additional arguments as determined by the programmer who creates the subroutine.

2. Include a statement at the beginning of the subroutine that reads the call number (first argument) and branches to a corresponding segment. Each segment processes the arguments from one call. For example, number one branches to the first segment, number two to the second segment, and so on.

3. Have each segment store the arguments it receives in other variables (which can be processed by the last segment) or accumulate them in a running total.

   End each segment with a command returning control back to the request (RETURN command).

4. The last segment returns the final output value to the request.

You can also use the entry point technique to write subroutines that process more than 200 arguments. For details, see *Executing a Subroutine at an Entry Point* on page 406.
**Syntax:** How to Create a Subroutine With Multiple Call Statements

```plaintext
field = subroutine (1, group1, field) ;
field = subroutine (2, group2, field);
.
.
.outfield = subroutine (n, groupn, outfield);
```

where:

- **field**
  
  Is the name of the field that contains the result of the segment or the format of the field enclosed in single quotation marks. This field must have the same format as outfield.
  
  Do not specify field for the last call statement; use outfield.

- **subroutine**
  
  Is the name of the subroutine up to eight characters long.

- **n**
  
  Is a number that identifies each subroutine call. It must be the first argument in each subroutine call. The subroutine uses this call number to branch to segments of code.

- **group1, group2,**...
  
  Are lists of input arguments passed by each subroutine call. Each group contains the same number of arguments, and no more than 26 arguments each.
  
  The final group may contain dummy arguments.

- **outfield**
  
  Is the field that contains the result, or the format of the output value enclosed in single quotation marks.
  
  In Dialogue Manager, you must specify the format. In Maintain, you must specify the name of the field.

**Example:** Creating a Subroutine Divided Into Segments

The ADD32 subroutine, written in pseudocode, sums 32 numbers. It is divided into six segments, each of which adds six numbers from a subroutine call. (The total number of input arguments is 36 but the last four are dummy arguments.) The sixth segment adds two arguments to the SUM variable and returns the result. The sixth segment does not process any values supplied for the four dummy arguments.
The subroutine is:

```
Subroutine ADD32 (NUM, A, B, C, D, E, F, TOTAL).
If NUM is 1 then goto label ONE
else if NUM is 2 then goto label TWO
else if NUM is 3 then goto label THREE
else if NUM is 4 then goto label FOUR
else if NUM is 5 then goto label FIVE
else goto label SIX.

Label ONE.
Let SUM = A + B + C + D + E + F.
Return.

Label TWO
Let SUM = SUM + A + B + C + D + E + F
Return

Label THREE
Let SUM = SUM + A + B + C + D + E + F
Return

Label FOUR
Let SUM = SUM + A + B + C + D + E + F
Return

Label FIVE
Let SUM = SUM + A + B + C + D + E + F
Return

Label SIX
LET TOTAL = SUM + A + B
Return
End of subroutine
```

To use the ADD32 subroutine, list all six call statements, each call specifying six numbers. The last four numbers, represented by zeros, are dummy arguments. The DEFINE command stores the total of the 32 numbers in the SUM32 field.

```
DEFINE FILE EMPLOYEE
DUMMY/D10 = ADD32 (1, 5, 7, 13, 9, 4, 2, DUMMY);
DUMMY/D10 = ADD32 (2, 5, 16, 2, 9, 28, 3, DUMMY);
DUMMY/D10 = ADD32 (3, 17, 12, 8, 4, 29, 6, DUMMY);
DUMMY/D10 = ADD32 (4, 28, 3, 22, 7, 18, 1, DUMMY);
DUMMY/D10 = ADD32 (5, 8, 19, 7, 25, 15, 4, DUMMY);
SUM32/D10 = ADD32 (6, 3, 27, 0, 0, 0, 0, SUM32);
END
```
Compiling and Storing a Subroutine

After you write a subroutine, you need to compile and store it. This topic discusses compiling and storing your subroutine for z/OS.

Compiling and Storing a Subroutine on z/OS

Compile the subroutine, then link-edit it and store the module in a load library. If your subroutine calls other subroutines, compile and link-edit all the subroutines together in a single module. Do not store the subroutine in the FUSELIB load library (FUSELIB.LOAD), as it may be overwritten when your site installs the next release of FOCUS.

If the subroutine is written in PL/I, include the following when link-editing the subroutine

ENTRY subroutine

where:

subroutine

Is the name of the subroutine.

Testing the Subroutine

How to:

Determine the Location of Error

After compiling and storing a subroutine, you can test it in a report request. In order to access the subroutine, you need to issue the ALLOCATE command for z/OS.

If an error occurs during testing, check to see if the error is in the request or in the subroutine.

Procedure: How to Determine the Location of Error

You can determine the location of an error with the following:

1. Write a dummy subroutine that has the same arguments but returns a constant.
2. Execute the request with the dummy subroutine.

If the request executes the dummy subroutine normally, the error is in your subroutine. If the request still generates an error, the error is in the request.
Using a Custom Subroutine: The MTHNAM Subroutine

In this section:
Writing the MTHNAM Subroutine
Calling the MTHNAM Subroutine From a Request

This topic discusses the MTHNAM subroutine as an example. The MTHNAM subroutine converts a number representing a month to the full name of that month. The subroutine processes as follows:

1. Receives the input argument from the request as a double-precision number.
2. Adds .000001 to the number which compensates for rounding errors. Rounding errors can occur since floating-point numbers are approximations and may be inaccurate in the last significant digit.
3. Moves the number into an integer field.
4. If the number is less than one or greater than 12, it changes the number to 13.
5. Defines a list containing the names of months and an error message for the number 13.
6. Sets the index of the list equal to the number in the integer field. It then places the corresponding array element into the output argument. If the number is 13, the argument contains the error message.
7. Returns the result as an output field.

Writing the MTHNAM Subroutine

Reference:
MTHNAM Subroutine Written in FORTRAN
MTHNAM Subroutine Written in COBOL
MTHNAM Subroutine Written in PL/I
MTHNAM Subroutine Written in BAL Assembler
MTHNAM Subroutine Written in C

The MTHNAM subroutine can be written in FORTRAN, COBOL, PL/I, BAL Assembler, and C.
**Reference:** MTHNAM Subroutine Written in FORTRAN

This is a FORTRAN version of the MTHNAM subroutine where:

**MTH**

Is the double-precision number in the input argument.

**MONTH**

Is the name of the month. Since the character string 'September' contains nine letters, MONTH is a three element array. The subroutine passes the three elements back to your application which concatenates them into one field.

**A**

Is a two dimensional, 13 by three array containing the names of the months. The last three elements contain the error message.

**IMTH**

Is the integer representing the month.

The subroutine is:

```fortran
SUBROUTINE MTHNAM (MTH, MONTH)
REAL*8     MTH
INTEGER*4  MONTH(3), A(13,3), IMTH
DATA
+     A( 1,1)/'JANU'/, A( 1,2)/'ARY '/, A( 1,3)/'    '/,
+     A(2,1)/'FEBR'/, A(2,2)/'UARY '/, A(2,3)/'    '/,
+     A(3,1)/'MARC'/, A(3,2)/'H '/, A(3,3)/'    '/,
+     A(4,1)/'APRI'/, A(4,2)/'L '/, A(4,3)/'    '/,
+     A(5,1)/'MAY '/, A(5,2)/' ', A(5,3)/'    '/,
+     A(6,1)/'JUNE'/, A(6,2)/' ', A(6,3)/'    '/,
+     A(7,1)/'JULY'/, A(7,2)/' ', A(7,3)/'    '/,
+     A(8,1)/'AUGU'/, A(8,2)/'ST '/, A(8,3)/'    '/,
+     A(9,1)/'SEPT'/, A(9,2)/'EMBE'/, A(9,3)/'R '/,
+     A(10,1)/'OCTO'/, A(10,2)/'BER '/, A(10,3)/'    '/,
+     A(11,1)/'NOVE'/, A(11,2)/'MBER'/, A(11,3)/'    '/,
+     A(12,1)/'DECE'/, A(12,2)/'MBER'/, A(12,3)/'    '/,
+     A(13,1)/'**ER'/, A(13,2)/'ROR*/', A(13,3)/'*   '/
IMTH=MTH+0.000001
IMTH=IMTH+0.000001
IF (IMTH .LT. 1 .OR. IMTH .GT. 12) IMTH=13
DO 1 I=1,3
1    MONTH(I)=A(IMTH,I)
RETURN
END
```

Using Functions 413
**Example:**  **Compiling the FORTRAN Version of MTHNAM Under LE on z/OS**

The following example compiles and link edits the FORTRAN version of MTHNAM:

```fortran
//COMPILE   EXEC  PGM=FORTVS2,
//          PARM='LANGLVL(66),NODECK,NOLIST,OPT(0)' /* PARM='NODECK,NOLIST,OPT(0)' */
//STEPLIB   DD  DSN=VSF2.VSF2COMP,DISP=SHR
//SYSLIB    DD  DSN=CEE.SCEESAMP,DISP=SHR
//SYSPRINT  DD  SYSOUT=*,DCB=BLKSIZE=3429
//SYSTEM   DD  SYSOUT=* 
//SYSPUNCH  DD  SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN    DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSD SPACE=(3200,(25,6)),DCB=BLKSIZE=3200
//SYSLIN    DD  DSN=&&LOADSET,DISP=(MOD,PASS)
//SYSLMOD   DD  DSN=prefix.TSO.LOAD,DISP=SHR
//OBJECT    DD  DSN=&&LOADSET,DISP=(OLD,DELETE) /* DD DDNAME=SYSIN */
//SYSLIN    DD  * INCLUDE OBJECT
//          INCLUDE SYSLIB(CEESG007)
//          NAME     MTHNAM(R)
/**
where:

**prefix**

Is the high-level qualifier for your production FOCUS data sets.
Reference: **MTHNAM Subroutine Written in COBOL**

This is a COBOL version of the MTHNAM subroutine where:

**MONTH-TABLE**

Is a field containing the names of the months and the error message.

**MLINE**

Is a 13-element array that redefines the MONTH-TABLE field. Each element (called A) contains the name of a month; the last element contains the error message.

**A**

Is one element in the MLINE array.

**IX**

Is an integer field that indexes MLINE.

**IMTH**

Is the integer representing the month.

**MTH**

Is the double-precision number in the input argument.

**MONTH**

Is the name of the month corresponding to the integer in IMTH.
The subroutine is:

IDENTIFICATION DIVISION.
PROGRAM-ID. MTHNAM.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.
DATA DIVISION.
WORKING-Storage SECTION.
  01 MONTH-TABLE.
    05 FILLER PIC X(9) VALUE 'JANUARY '.
    05 FILLER PIC X(9) VALUE 'FEBRUARY '.
    05 FILLER PIC X(9) VALUE 'MARCH '.
    05 FILLER PIC X(9) VALUE 'APRIL '.
    05 FILLER PIC X(9) VALUE 'MAY '.
    05 FILLER PIC X(9) VALUE 'JUNE '.
    05 FILLER PIC X(9) VALUE 'JULY '.
    05 FILLER PIC X(9) VALUE 'AUGUST '.
    05 FILLER PIC X(9) VALUE 'SEPTEMBER'.
    05 FILLER PIC X(9) VALUE 'OCTOBER '.
    05 FILLER PIC X(9) VALUE 'NOVEMBER '.
    05 FILLER PIC X(9) VALUE 'DECEMBER '.
    05 FILLER PIC X(9) VALUE '**ERROR**'.
  01 MLIST REDEFINES MONTH-TABLE.
    05 MLINE OCCURS 13 TIMES INDEXED BY IX.
      10 A  PIC X(9).
  01 IMTH    PIC S9(5) COMP.
LINKAGE SECTION.
  01 MTH     COMP-2.
  01 MONTH   PIC X(9).
PROCEDURE DIVISION USING MTH, MONTH.
BEG-1.
  ADD 0.000001 TO MTH.
  MOVE MTH TO IMTH.
  IF IMTH < +1 OR > 12
    SET IX TO +13
  ELSE
    SET IX TO IMTH.
  MOVE A (IX) TO MONTH.
GOBACK.
Example: Compiling the COBOL Version of MTHNAM Under LE on z/OS

The following example compiles and linkedit the COBOL version of MTHNAM:

```cobol
//COMPILE   EXEC  PGM=IGYCRCTL,
//          PARM='APOST,RES,RENT'
//STEPLIB   DD  DSN=IGY.V1R2M0.SIGYCOMP,DISP=SHR

//SYSPRINT DD SYSOUT=* 
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSDA,DISP=(MOD,PASS),
//          SPACE=(TRK,(3,3))
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSIN    DD *
/*
/*  The subroutine source code goes here
/*  Alternatively, your DD statement can point to a data set
/*     that contains the source code.
/*
/**
//LINKEDIT EXEC  PGM=IEWL,
//          PARM='REUS,MAP,LIST'
//STEPLIB DD DSN=CEE.SCEELKED,DISP=SHR
//OBJECT DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD DD SYSOUT=* 
//SYSLIN DD *
MODE AMODE(31),RMODE(ANY)
INCLUDE OBJECT
ENTRY MTHNAM
NAME MTHNAM(R)
/**

where:

prefix

Is the high-level qualifier for your production FOCUS data sets.

Note:

RENT is required for reentrancy.
The linkedit parameter REUSE must be used to make the load module reusable.

CEEUOPT is automatically included for compiles and linkeds.

**Reference:**  **MTHNAM Subroutine Written in PL/I**

This is a PL/I version of the MTHNAM subroutine where:

- **MTHNUM**
  - Is the double-precision number in the input argument.

- **FULLMTH**
  - Is the name of the month corresponding to the integer in MONTHNUM.

- **MONTHNUM**
  - Is the integer representing the month.

- **MONTH_TABLE**
  - Is a 13-element array containing the names of the months. The last element contains the error message.

The subroutine is:

```cobol
MTHNAM:  PROC(MTHNUM,FULLMTH) OPTIONS(COBOL);
DECLARE  MTHNUM  DECIMAL FLOAT (16) ;
DECLARE  FULLMTH CHARACTER (9) ;
DECLARE  MONTHNUM FIXED BIN (15,0)  STATIC ;
DECLARE  MONTH_TABLE(13) CHARACTER (9)   STATIC
  INIT  ('JANUARY',
      'FEBRUARY',
      'MARCH',
      'APRIL',
      'MAY',
      'JUNE',
      'JULY',
      'AUGUST',
      'SEPTEMBER',
      'OCTOBER',
      'NOVEMBER',
      'DECEMBER',
      '**ERROR**') ;

MONTHNUM = MTHNUM + 0.00001 ;
IF MONTHNUM < 1 | MONTHNUM > 12 THEN
    MONTHNUM = 13 ; FULLMTH = MONTH_TABLE(MONTHNUM) ;
RETURN;
END MTHNAM;
```
Example: Compiling the PL/I Version of MTHNAM Under LE on z/OS

This example includes the following steps for compiling, linkediting, and calling the PL/I version of MTHNAM:

1. Compiling the COBOL stub:

```cobol
/* Step 1 - compile the COBOL stub

//COBSTUB EXEC IGYWCL,
//  PARM.COBOL='APOST,DYNAM,RENT',
//  PARM.LKED='LIST,MAP,SIZE=2046K'
//COBOL.SYSIN     DD  *
IDENTIFICATION DIVISION.
PROGRAM-ID. COBSTUB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.
DATA DIVISION.
WORKING-Storage SECTION.
LINKAGE SECTION.
  01 DUMMY-IO PIC X(99).
PROCEDURE DIVISION USING DUMMY-IO.
MAIN SECTION.
  CALL 'MTHNAM' USING DUMMY-IO.
MAIN-EXIT. EXIT.
  GOBACK.

/*
/**
//LKED.SYSIN DD *
  MODE AMODE(31),RMODE(ANY)
  ENTRY COBSTUB
  NAME COBSTUB(R)
/**
//LKED.SYSLMOD DD DSN=prefix.TSO.LOAD,DISP=SHR
/**
```
2. Compiling the PL/I subroutine:

        //* Step 2 - compile the PLI program
        //*
        //*COMPILE EXEC IEL1CL,
        //     PARM.PLI='OBJECT,NODECK',
        //     PARM.LKED='XREF,LIST'
        //*
        //PLI.SYIN DD *
        /*
        /* The subroutine source code goes here.
        /* Alternatively, your DD statement can point to the data set
        /* containing the source code.
        /*
        //LKED.SYSLMOD DD DSN=prefix.TSO.LOAD,DISP=SHR
        //LKED.SYIN DD *
        ENTRY MTHNAM
        NAME MTHNAM(R)
        /*
        //*
A. Creating a Subroutine

3. Executing FOCUS:

```plaintext
//FOCUS EXEC PGM=FOCUS,COND=(0,NE)
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
  // DD DSN=prefix.TSO.LOAD,DISP=SHR
  // DD DSN=prefix.FOCLIB.LOAD,DISP=SHR
  // DD DSN=prefix.FUSELIB.LOAD,DISP=SHR
//USERLIB DD DSN=prefix.FOCLIB.LOAD,DISP=SHR
//ERRORS DD DSN=prefix.ERRORS.DATA,DISP=SHR
//MASTER DD DSN=prefix.MASTER.DATA,DISP=SHR
//FOCEXEC DD DSN=prefix.FOCEXEC.DATA,DISP=SHR
//EMPLOYEE DD DSN=prefix.EMPLOYEE.FOCUS,DISP=SHR
  // SYSOUT DD SYSOUT=*  
  // SYSPRINT DD SYSOUT=*  
  // FSTRACE DD SYSOUT=*  
  // OUT DD SYSOUT=*,DCB=BLKSIZE=121  
  // OFFLINE DD SYSOUT=*  
  // * SYSUDUMP DD DUMMY  
  // SYSIN DD *  
SET PRINT = OFFLINE  
DEFINE FILE EMPLOYEE  
  MONTH_NUM/M = PAY_DATE;  
  PAY_MONTH/A12 = MTHNAM (MONTH_NUM, PAY_MONTH);  
END  
TABLE FILE EMPLOYEE  
  HEADING  
    " "  
    "FOCUS RELEASE - &FOCREL  PUT LEVEL - " &PUTLEVEL  
    "SUBROUTINE - MTHNAM "  
    " "  
  PRINT PAY_MONTH GROSS  
  BY EMP_ID BY FIRST_NAME BY LAST_NAME  
  BY PAY_DATE  
  IF LAST_NAME IS STEVENS  
END  
FIN  
/*  
/
```

where:

`prefix`

Is the high-level qualifier for your production FOCUS data sets.

Using Functions
Using a Custom Subroutine: The MTHNAM Subroutine

Reference: MTHNAM Subroutine Written in BAL Assembler

This is a BAL Assembler version of the MTHNAM subroutine:

* =====================================================================
* A SIMPLE MAIN ASSEMBLE ROUTINE THAT CALLS THE LE CALLABLE SERVICES
* =====================================================================

MTHNAM   CEEENTRY PPA=MAINPPA,AUTO=WORKSIZE,MAIN=NO
USING WORKAREA,13

L        3,0(0,1)          LOAD ADDR OF FIRST ARG  INTO R3
LD       4,=D'0.0'         CLEAR OUT FPR4 AND FPR5
LE       6,0(0,3)          FP NUMBER IN FPR6
LPER     4,6               ABS VALUE IN FPR4
AW       4,=D'0.00001'     ADD ROUNDING CONSTANT
AW       4,DZERO           SHIFT OUT FRACTION
STD      4,FPNUM           MOVE TO MEMORY
L        2,FPNUM+4         INTEGER PART IN R2
TM       0(3),B'10000000'  CHECK SIGN OF ORIGINAL NO
BNO      POS               BRANCH IF POSITIVE
LCR      2,2               COMPLEMENT IF NEGATIVE

POS      LR       3,2              COPY MONTH NUMBER INTO R3
C        2,=F'0'          IS IT ZERO OR LESS?
BNP      INVALID          YES. SO INVALID
C        2,=F'12'         IS IT GREATER THAN 12?
BNP      VALID           NO. SO VALID
INVALID  LA       3,13(0,0)        SET R3 TO POINT TO ITEM 13 (ERROR)

VALID    SR       2,2               CLEAR OUT R2
M        2,=F'9'         MULTIPLY BY SHIFT IN TABLE

LA       6,MTH(3)          GET ADDR OF ITEM IN R6
L        4,4(0,1)         GET ADDR OF SECOND ARG IN R4
MVC      0(9,4),0(6)     MOVE IN TEXT

* TERMINATE THE CEE ENVIRONMENT AND RETURN TO THE CALLER
* CEETERM  RC=0
A. Creating a Subroutine

*  ================================================================
*  CONSTANTS
*  ================================================================
  DS    0D                  ALIGNMENT
  FPNUM DS    D                   FLOATING POINT NUMBER
  DZERO DC X'4E00000000000000' SHIFT CONSTANT
  MTH   DC CL9'DUMMYITEM'      MONTH TABLE
        DC CL9'JANUARY'
        DC CL9'FEBRUARY'
        DC CL9'MARCH'
        DC CL9'APRIL'
        DC CL9'MAY'
        DC CL9'JUNE'
        DC CL9'JULY'
        DC CL9'AUGUST'
        DC CL9'SEPTMBER'
        DC CL9'OCTOBER'
        DC CL9'NOVEMBER'
        DC CL9'DECEMBER'
        DC CL9'**ERROR**'

  * MAINPPA  CEEPPA                  CONSTANTS DESCRIBING THE CODE BLOCK
  *  ================================================================
*  THE WORKAREA AND DSA
*  ================================================================

WORKAREA DSECT
  ORG    *+CEEDSASZ        LEAVE SPACE FOR THE DSA FIXED PART
  PLIST  DS    0D
  PARM1  DS    A
  PARM2  DS    A
  PARM3  DS    A
  PARM4  DS    A
  PARM5  DS    A

  * FOCPARM1 DS    F                 SAVE FIRST PARAMETER PASSED
  FOCPARM2 DS    F                 SAVE SECOND PARAMETER PASSED
  *
  * DS    0D
  WORKSIZE EQU    *-WORKAREA
  CEEDSA                  MAPPING OF THE DYNAMIC SAVE AREA
  CEECAA                  MAPPING OF THE COMMON ANCHOR AREA

  *  END   MTHNAM                 NOMINATE MTHNAM AS THE ENTRY POINT
/**
**Example:**  **Assemble the BAL Version of MTHNAM Under LE on z/OS**

The following example assembles and linkedits the Assembler version of MTHNAM:

```plaintext
//ASSEMBLE  EXEC  PGM=ASMA90,
  //          PARM='OBJECT,LIST,ESD,NODECK'
//SYSLIB    DD  DSN=CEE.SCEEMAC,DISP=SHR
//SYSUT1    DD  UNIT=SYSDA,SPACE=(TRK,(45,15))
//SYSUT2    DD  UNIT=SYSDA,SPACE=(TRK,(45,15))
//SYSUT3    DD  UNIT=SYSDA,SPACE=(TRK,(45,15))
//SYSPUNCH  DD  DUMMY
//SYSPRINT  DD  SYSOUT=*  
//SYSLIN    DD  DSNAME=&&LINKSET,UNIT=SYSDA,DISP=(MOD,PASS),
  //          SPACE=(TRK,(3,3))
//SYSIN     DD  *

/*
/* The subroutine source code goes here.
/* Alternatively, the DD statement can point to a data set that contains
/* the source code.
*/

/*
//IEBGENER  EXEC  PGM=IEBGENER,
  //          COND=(0,NE)
//SYSUT1    DD  DSN=&&LINKSET,DISP=(OLD,PASS)
//SYSUT2    DD  SYSOUT=*  
//SYSPRINT  DD  DUMMY
//SYSIN     DD  DUMMY

/*
//LINKEDIT  EXEC  PGM=IEWL,
  //          PARM='LIST,XREF,LET,REUS',
  //          COND=(0,NE)
//SYSLIB    DD  DSN=CEE.SCEELKED,DISP=SHR
//SYSUT1    DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD   DD  DSN=prefix.TSO.LOAD(MTHNAM),DISP=SHR
//SYSPRINT  DD  SYSOUT=*  
//SYSLIN    DD  DSNAME=&&LINKSET,DISP=(OLD,PASS)
//DDNAME=SYSIN

/*
//SYSIN     DD  *
  ENTRY MTHNAM
  NAME  MTHNAM(R)
/*

**where:**

`prefix`

Is the high-level qualifier for your production FOCUS data sets.
Reference: MTHNAM Subroutine Written in C

This is a C language version of the MTHNAM subroutine:

```c
void mthnam(double *,char *);
void mthnam(double *mth, char *month);

{ char *nmonth[13] = {
    "January ",
    "February ",
    "March ",
    "April ",
    "May ",
    "June ",
    "July ",
    "August ",
    "September",
    "October ",
    "November ",
    "December ",
    "**Error**"};

    int imth, loop;
    imth = *mth + .00001;
    imth = (imth < 1 || imth > 12 ? 13 : imth);
    for (loop=0;loop < 9;loop++)
        month[loop] = nmonth[imth-1][loop];
}
```
Example: Compiling the C Version of MTHNAM Under LE on z/OS

The following example compiles and linkedits the C version of MTHNAM:

```
//CBG     EXEC PROC=EDCCL
//COMPILE.SYSPRINT DD SYSOUT=*,
//                   DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//COMPILE.SYSIN     DD  *,DLM=XX
/*  #INCLUDE <STDIO.H> */
/*
/* The subroutine source code goes here.
/* Alternatively, the DD statement can point to a data set that contains
/* the source code.
/*
//LKED.SYSPRINT  DD  SYSOUT=*  
//LKED.SYSLMOD   DD  DSN=prefix.LOADLIB,DISP=SHR
//LKED.SYSIN  DD  *  
     NAME MTHNAM(R)
/*
/*
```

where:

prefix

Is the high-level qualifier for your production FOCUS data sets.

Note:

- A STEPLIB and/or SYSLIB may be required to access C functions.
- The Language Environment has been enhanced to load the required libraries for LE-C upon entering FOCUS.

Calling the MTHNAM Subroutine From a Request

You can call the MTHNAM subroutine from a report request.
**Example:** Calling the MTHNAM Subroutine

The DEFINE command extracts the month portion of the pay date. The MTHNAM subroutine then converts it into the full name of the month, and stores the name in the PAY_MONTH field. The report request prints the monthly pay of Alfred Stevens.

```plaintext
DEFINE FILE EMPLOYEE
MONTH_NUM/M = PAY_DATE;
PAY_MONTH/A12 = MTHNAM (MONTH_NUM, PAY_MONTH);
END
TABLE FILE EMPLOYEE
PRINT PAY_MONTH GROSS
BY EMP_ID BY FIRST NAME BY LAST_NAME
BY PAY_DATE
IF LN IS STEVENS
END
```

The output is:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PAY_DATE</th>
<th>PAY_MONTH</th>
<th>GROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>071382660</td>
<td>ALFRED</td>
<td>STEVENS</td>
<td>81/11/30</td>
<td>NOVEMBER</td>
<td>$833.33</td>
</tr>
<tr>
<td>81/12/31</td>
<td></td>
<td></td>
<td></td>
<td>DECEMBER</td>
<td>$833.33</td>
</tr>
<tr>
<td>82/01/29</td>
<td></td>
<td></td>
<td></td>
<td>JANUARY</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/02/26</td>
<td></td>
<td></td>
<td></td>
<td>FEBRUARY</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/03/31</td>
<td></td>
<td></td>
<td></td>
<td>MARCH</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/04/30</td>
<td></td>
<td></td>
<td></td>
<td>APRIL</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/05/28</td>
<td></td>
<td></td>
<td></td>
<td>MAY</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/06/30</td>
<td></td>
<td></td>
<td></td>
<td>JUNE</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/07/30</td>
<td></td>
<td></td>
<td></td>
<td>JULY</td>
<td>$916.67</td>
</tr>
<tr>
<td>82/08/31</td>
<td></td>
<td></td>
<td></td>
<td>AUGUST</td>
<td>$916.67</td>
</tr>
</tbody>
</table>

**Subroutines Written in REXX**

In this section:
- Formats and REXX Subroutines

How to:
- Allocate REXX Subroutines
- Call a REXX Subroutine

A request can call a subroutine coded in REXX. These subroutines, also called FUSREXX macros, provide a 4GL option to the languages supported for user-written subroutines.

REXX subroutines are supported in the z/OS environment. A REXX subroutine contains REXX source code. Compiled REXX code is not supported.

REXX subroutines are not necessarily the same in all operating environments. Therefore, some of the examples may use REXX functions that are not available in your environment.
Because of CPU requirements, the use of REXX subroutines in large production jobs should be monitored carefully.

For more information on REXX subroutines, see your REXX documentation.

**Syntax:**

**How to Allocate REXX Subroutines**

REXX subroutines must be stored as members of a PDS. By default, REXX subroutines are searched for in the data sets allocated to DDNAME FUSREXX and then using the standard z/OS search order.

By issuing the SET FUSREXXDD command, you change the search path to search first in the SYSEXEC allocation and then use the standard z/OS search order. The syntax is:

```plaintext
SET FUSREXXDD = {ON|OFF}
```

where:

- **ON**
  - Specifies that REXX subroutines will be found in the allocations for DDNAME FUSREXX. ON is the default value.

- **OFF**
  - Specifies that REXX subroutines will be found in the SYSEXEC allocations.

**Note:** Once you call a REXX subroutine, the FUSREXXDD parameter value cannot be changed for the remainder of the session.

The search order for a REXX subroutine is:

1. FUSREXX or SYSEXEC, depending on the FUSREXXDD parameter value.
2. Standard z/OS search order.
**Syntax:**

**How to Call a REXX Subroutine**

```
DEFINE FILE filename
fieldname/{An|In} = subname(inlen1, inparml, ..., outlen, outparm);
END

or

{DEFINE|COMPUTE} fieldname/{An|In} = subname(inlen1, inparml, ..., outlen, outparm);

or

-SET &var = subname(inlen1, inparml, ..., outlen, outparm);
```

where:

- **fieldname**
  - Is the field that contains the result.
- **An, In**
  - Is the format of the field that contains the result.
- **subname**
  - Is the name of the REXX subroutine.
- **inlen1, inparml ...**
  - Are the input parameters. Each parameter consists of a length and an alphanumeric parameter value. You can supply the value, the name of an alphanumeric field that contains the value, or an expression that returns the value. Up to 13 input parameter pairs are supported. Each parameter value can be up to 256 bytes long.
  
  Dialogue Manager converts numeric arguments to floating-point double-precision format. Therefore, you can only pass alphanumeric input parameters to a REXX subroutine using -SET.

- **outlen, outparm**
  - Is the output parameter pair, consisting of a length and a result. In most cases, the result should be alphanumeric, but integer results are also supported. The result can be a field or a Dialogue Manager variable that contains the value, or the format of the value enclosed in single quotation marks. The return value can be a minimum of one byte long and a maximum (for an alphanumeric value) of 256 bytes.

**Note:** If the value returned is an integer, **outlen** must be 4 because FOCUS reserves four bytes for integer fields.

- **&var**
  - Is the name of the Dialogue Manager variable that contains the result.
**Example:**  Returning the Day of the Week

The REXX subroutine DOW returns the day of the week corresponding to the date an employee was hired. The routine contains one input parameter pair and one return field pair.

```rexx
DEFINE FILE EMPLOYEE
1. AHDT/A6 = EDIT(HIRE_DATE) ;
2. DAY_OF_WEEK/A9 WITH AHDT = DOW(6, AHDT, 9, DAY_OF_WEEK);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME HIRE_DATE DAY_OF_WEEK
END
```

The procedure processes as follows:

1. The EDIT function converts HIRE_DATE to alphanumeric format and stores the result in a field with the format A6.

2. The result is stored in the DAY_OF_THE_WEEK field, and can be up to nine bytes long.

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>DAY_OF_WEEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>80/06/02</td>
<td>Monday</td>
</tr>
<tr>
<td>SMITH</td>
<td>81/07/01</td>
<td>Wednesday</td>
</tr>
<tr>
<td>JONES</td>
<td>82/05/01</td>
<td>Saturday</td>
</tr>
<tr>
<td>SMITH</td>
<td>82/01/04</td>
<td>Monday</td>
</tr>
<tr>
<td>BANNING</td>
<td>82/08/01</td>
<td>Sunday</td>
</tr>
<tr>
<td>IRVING</td>
<td>82/01/04</td>
<td>Monday</td>
</tr>
<tr>
<td>ROMANS</td>
<td>82/07/01</td>
<td>Thursday</td>
</tr>
<tr>
<td>MCCOY</td>
<td>81/07/01</td>
<td>Wednesday</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>82/04/01</td>
<td>Thursday</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>82/02/02</td>
<td>Tuesday</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>82/04/01</td>
<td>Thursday</td>
</tr>
<tr>
<td>CROSS</td>
<td>81/11/02</td>
<td>Monday</td>
</tr>
</tbody>
</table>

The REXX subroutine appears below. It reads the input date, reformats it to MM/DD/YY format, and returns the day of the week using a REXX DATE call.

```rexx
/* DOW routine. Return WEEKDAY from YYMMDD format date */
Arg ymd .
Return Date('W',Translate('34/56/12',ymd,'123456'),'U')
```
**Example:** Passing Multiple Arguments to a REXX Subroutine

The REXX subroutine INTEREST has four input parameters.

```rexx
DEFINE FILE EMPLOYEE
1. AHDT/A6     = EDIT(HIRE_DATE);
2. ACSAL/A12   = EDIT(CURR_SAL);
3. DCSAL/D12.2 = CURR_SAL;
4. PV/A12    = INTEREST(6, AHDT, 6, 'YMD', 3, '6.5', 12, ACSAL, 12, PV);
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME HIRE_DATE DCSAL PV
END
```

The procedure processes as follows:

1. **EDIT** converts HIRE_DATE to alphanumeric format and stores the result in AHDT.
2. **EDIT** converts CURR_SAL to alphanumeric format and stores the result in ACSAL.
3. CURR_SAL is converted to a floating-point double-precision field that includes commas, and the result is stored in DCSAL.
4. The second input field is six bytes long. Data is passed as a character variable &YMD in YYMMDD format.
   - The third input field is a character value of 6.5, which is three bytes long to account for the decimal point in the character string.
   - The fourth input field is 12 bytes long. This passes the character field ACSAL.
   - The return field is up to 12 bytes long and is named PV.

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>DCSAL</th>
<th>PV</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>11,000.00</td>
<td>14055.14</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>13,200.00</td>
<td>15939.99</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>18,480.00</td>
<td>21315.54</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>9,500.00</td>
<td>11155.60</td>
</tr>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>29,700.00</td>
<td>33770.53</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>26,862.00</td>
<td>31543.35</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>21,120.00</td>
<td>24131.19</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>18,480.00</td>
<td>22315.99</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>21,780.00</td>
<td>25238.25</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>16,100.00</td>
<td>18822.66</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>9,000.00</td>
<td>10429.03</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>27,062.00</td>
<td>32081.82</td>
</tr>
</tbody>
</table>

Using Functions 431
The REXX subroutine appears below. The REXX Format command is used to format the return value.

```rexx
/* Simple INTEREST program. dates are yymmdd format */
Arg start_date,now_date,percent,open_balance, .

begin = Date('B',Translate('34/56/12',start_date,'123456'),'U')
stop  = Date('B',Translate('34/56/12',now_date,'123456'),'U')
valnow = open_balance * (((stop - begin) * (percent / 100)) / 365)

Return Format(valnow,9,2)
```

**Example: Accepting Multiple Tokens in a Parameter**

A REXX subroutine can accept multiple tokens in a parameter. The following procedure passes employee information (PAY_DATE and MO_PAY) as separate tokens in the first parameter. It passes three input parameters and one return field.

```rexx
DEFINE FILE EMPLOYEE
1. COMPID/A256 = FN | ' ' | LN | ' ' | DPT | ' ' | EID ;
2. APD/A6 = EDIT(PAY_DATE);
3. APAY/A12 = EDIT(MO_PAY);
4. OK4RAISE/A1 = OK4RAISE(256, COMPID, 6, APD, 12, APAY, 1, OK4RAISE);
END

TABLE FILE EMPLOYEE
PRINT EMP_ID FIRST_NAME LAST_NAME DEPARTMENT
IF OK4RAISE EQ '1'
END
```

The procedure processes as follows:

1. **COMPID** is the concatenation of several character fields passed as the first parameter and stored in a field with the format A256. Each of the other parameters is a single argument.

2. **EDIT** converts PAY_DATE to alphanumeric format.

3. **EDIT** converts MO_PAY to alphanumeric format.

4. **OK4RAISE** executes, and the result is stored in OK4RAISE.

The output is:

```
EMP_ID   FIRST_NAME   LAST_NAME   DEPARTMENT
-------   ----------   ---------   ----------
071382660 ALFRED      STEVENS     PRODUCTION
```

Subroutines Written in REXX
The REXX subroutine appears below. Commas separate FUSREXX parameters. The ARG command specifies multiple variable names before the first comma and, therefore, separates the first FUSREXX parameter into separate REXX variables, using blanks as delimiters between the variables.

/* OK4RAISE routine. Parse separate tokens in the 1st parm, */ /* then more parms */

Arg fname lname dept empid, pay_date, gross_pay, .

If dept = 'PRODUCTION' & pay_date < '820000'
Then retvalue = '1'
Else retvalue = '0'

Return retvalue

REXX subroutines should use the REXX RETURN subroutine to return data. REXX EXIT is acceptable, but is generally used to end an EXEC, not a FUNCTION.

<table>
<thead>
<tr>
<th>Correct</th>
<th>Not as Clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Some FUSREXX function */</td>
<td>/* Another FUSREXX function */</td>
</tr>
<tr>
<td>Arg input</td>
<td>Arg input</td>
</tr>
<tr>
<td>some rexx process ...</td>
<td>some rexx process ...</td>
</tr>
<tr>
<td>Return data_to_WebFOCUS</td>
<td>Exit 0</td>
</tr>
</tbody>
</table>

**Formats and REXX Subroutines**

A REXX subroutine requires input data to be in alphanumeric format. Most output is returned in alphanumeric format. If the format of an input argument is numeric, use the EDIT or FTOA functions to convert the argument to alphanumeric. You can then use the EDIT or ATODBL functions to convert the output back to numeric.

The output length in the subroutine call must be four. Character variables cannot be more than 256 bytes. This limit also applies to REXX subroutines. FUSREXX routines return variable length data. For this reason, you must supply the length of the input arguments and the maximum length of the output data.

A REXX subroutine does not require any input parameters, but requires one return parameter, which must return at least one byte of data. It is possible for a REXX subroutine not to need input, such as a function that returns USERID.
A REXX subroutine does not support FOCUS date input arguments. When working with dates you can do one of the following:

- Pass an alphanumerical field with date display options and have the subroutine return a date value.

Date fields contain the integer number of days since the base date 12/31/1900. REXX has a date function that can accept and return several types of date formats, including one called Base format (‘B’) that contains the number of days since the REXX base date 01/01/0001. You must account for the number of days difference between the FOCUS base date and the REXX base date and convert the result to integer.

- Pass a date value converted to alphanumerical format. You must account for the difference in base dates for both the input and output arguments.

**Example:** Returning a Result in Alphanumeric Format

The NUMCNT subroutine returns the number of copies of each classic movie in alphanumerical format. It passes one input parameter and one return field.

```
TABLE FILE MOVIES
PRINT TITLE AND COMPUTE
1. ACOPIES/A3 = EDIT(COPIES); AS 'COPIES'
   AND COMPUTE
2. TXTCOPIES/A8 = NUMCNT(3, ACOPIES, 8, TXTCOPIES);
   WHERE CATEGORY EQ 'CLASSIC'
END
```

The procedure processes as follows:

1. The EDIT field converts COPIES to alphanumerical format, and stores the result in ACOPIES.
2. The result is stored in an 8-byte alphanumerical field TXTCOPIES.

The output is:

```
<table>
<thead>
<tr>
<th>TITLE</th>
<th>COPIES</th>
<th>TXTCOPIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAST OF EDEN</td>
<td>001</td>
<td>One</td>
</tr>
<tr>
<td>CITIZEN KANE</td>
<td>003</td>
<td>Three</td>
</tr>
<tr>
<td>CYRANO DE BERGERAC</td>
<td>001</td>
<td>One</td>
</tr>
<tr>
<td>MARTY</td>
<td>001</td>
<td>One</td>
</tr>
<tr>
<td>MALTESE FALCON, THE</td>
<td>002</td>
<td>Two</td>
</tr>
<tr>
<td>GONE WITH THE WIND</td>
<td>003</td>
<td>Three</td>
</tr>
<tr>
<td>ON THE WATERFRONT</td>
<td>002</td>
<td>Two</td>
</tr>
<tr>
<td>MUTINY ON THE BOUNTY</td>
<td>002</td>
<td>Two</td>
</tr>
<tr>
<td>PHILADELPHIA STORY, THE</td>
<td>002</td>
<td>Two</td>
</tr>
<tr>
<td>CAT ON A HOT TIN ROOF</td>
<td>002</td>
<td>Two</td>
</tr>
<tr>
<td>CASABLANCA</td>
<td>002</td>
<td>Two</td>
</tr>
</tbody>
</table>
```
The subroutine is:

/* NUMCNT routine. */
/* Pass a number from 0 to 10 and return a character value */
Arg numbr.
data = 'Zero One Two Three Four Five Six Seven Eight Nine Ten'
numbr = numbr + 1           /* so 0 equals 1 element in array */
Return Word(data,numbr)

Example: Returning a Result in Integer Format

In the following example, the NUMDAYS subroutine finds the number of days between
HIRE_DATE and DAT_INC and returns the result in integer format.

DEFINE FILE EMPLOYEE
1. AHDT/A6 = EDIT(HIRE_DATE);
2. ADI/A6 = EDIT(DAT_INC);
3. BETWEEN/I6 = NUMDAYS(6, AHDT, 6, ADI, 4, 'I6') ;
END

TABLE FILE EMPLOYEE
PRINT LAST_NAME HIRE_DATE DAT_INC BETWEEN
IF BETWEEN NE 0
END

The procedure processes as follows:

1. EDIT converts HIRE_DATE to alphanumeric format and stores the result in AHDT.
2. EDIT converts DAT_INC to alphanumeric format and stores the result in ADI.
3. NUMDAYS finds the number of days between AHDT and ADI and stores the result in
   integer format.

The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>DAT_INC</th>
<th>BETWEEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEVENS</td>
<td>80/06/02</td>
<td>82/01/01</td>
<td>578</td>
</tr>
<tr>
<td>STEVENS</td>
<td>80/06/02</td>
<td>81/01/01</td>
<td>213</td>
</tr>
<tr>
<td>SMITH</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
</tr>
<tr>
<td>JONES</td>
<td>82/05/01</td>
<td>82/06/01</td>
<td>31</td>
</tr>
<tr>
<td>SMITH</td>
<td>82/01/04</td>
<td>82/05/14</td>
<td>130</td>
</tr>
<tr>
<td>IRVING</td>
<td>82/01/04</td>
<td>82/05/14</td>
<td>130</td>
</tr>
<tr>
<td>MCCOY</td>
<td>81/07/01</td>
<td>82/01/01</td>
<td>184</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>82/02/02</td>
<td>82/05/14</td>
<td>101</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>82/04/01</td>
<td>82/06/11</td>
<td>71</td>
</tr>
<tr>
<td>CROSS</td>
<td>81/11/02</td>
<td>82/04/09</td>
<td>158</td>
</tr>
</tbody>
</table>
Subroutines Written in REXX

The subroutine appears below. The return value is converted from REXX character to HEX and formatted to be four bytes long.

/* NUMDAYS routine. */
/* Return number of days between 2 dates in yymmd format */
/* The value returned will be in hex format */

Arg first,second .

base1 = Date('B',Translate('34/56/12',first,'123456'),'U')
base2 = Date('B',Translate('34/56/12',second,'123456'),'U')

Return D2C(base2 - base1,4)

Example:  Passing a Date Value as an Alphanumeric Field With Date Options

In the following example, a date is used by passing an alphanumeric field with date options to the DATEREX1 subroutine. DATEREX1 takes two input arguments: an alphanumeric date in A8YYMD format and a number of days in character format. It returns a smart date in YYMD format that represents the input date plus the number of days. The format A8YYMD corresponds to the REXX Standard format ('S').

The number 693959 represents the number of days difference between the FOCUS base date and the REXX base date:

/* REXX DATEREX1 routine. Add indate (format A8YYMD) to days */
Arg indate, days .
Return D2C(Date('B',indate,'S')+ days - 693959, 4)

The following request uses the DATEREX1 macro to calculate the date that is 365 days from the hire date of each employee. The input arguments are the hire date and the number of days to add. Because HIRE_DATE is in I6YMD format, it must be converted to A8YYMD before being passed to the macro:

TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME HIRE_DATE
AND COMPUTE
  ADATE/YYMD = HIRE_DATE; NOPRINT
AND COMPUTE
  INDATE/A8YYMD= ADATE; NOPRINT
AND COMPUTE
  NEXT_DATE/YYMD = DATEREX1(8, INDATE, 3, '365', 4, NEXT_DATE);
BY LAST_NAME NOPRINT
END
The output is:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEXT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>1983/08/01</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1983/04/01</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1982/11/02</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1983/04/01</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>1983/01/04</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1983/05/01</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1982/07/01</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>1983/02/02</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>1983/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1982/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>1983/01/04</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>1981/06/02</td>
</tr>
</tbody>
</table>

**Example:** Passing a Date as a Date Converted to Alphanumeric Format

In the following example, a date is passed to the subroutine as a smart date converted to alphanumeric format. The DATEREX2 subroutine takes two input arguments: an alphanumeric number of days that represents a smart date, and a number of days to add. It returns a smart date in YYMD format that represents the input date plus the number of days. Both the input date and output date are in REXX base date ('B') format.

The number 693959 represents the number of days difference between the FOCUS base date and the REXX base date:

```
/* REXX DATEREX2 routine. Add indate (original format YYMD) to days */
Arg indate, days .
Return D2C(Date('B',indate+693959,'B') + days - 693959, 4)
```

The following request uses DATEREX2 to calculate the date that is 365 days from the hire date of each employee. The input arguments are the hire date and the number of days to add. Because HIRE_DATE is in I6YMD format, it must be converted to an alphanumeric number of days before being passed to the macro:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME FIRST_NAME HIRE_DATE
AND COMPUTE
  ADATE/YYMD = HIRE_DATE; NOPRINT
AND COMPUTE
  INDATE/A8 = EDIT(ADATE); NOPRINT
AND COMPUTE
  NEXT_DATE/YYMD = DATEREX2(8,INDATE,3,'365',4,NEXT_DATE);
BY LAST_NAME NOPRINT
END
```

Using Functions | 437
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>HIRE_DATE</th>
<th>NEXT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANNING</td>
<td>JOHN</td>
<td>82/08/01</td>
<td>1983/08/01</td>
</tr>
<tr>
<td>BLACKWOOD</td>
<td>ROSEMARIE</td>
<td>82/04/01</td>
<td>1983/04/01</td>
</tr>
<tr>
<td>CROSS</td>
<td>BARBARA</td>
<td>81/11/02</td>
<td>1982/11/02</td>
</tr>
<tr>
<td>GREENSPAN</td>
<td>MARY</td>
<td>82/04/01</td>
<td>1983/04/01</td>
</tr>
<tr>
<td>IRVING</td>
<td>JOAN</td>
<td>82/01/04</td>
<td>1983/01/04</td>
</tr>
<tr>
<td>JONES</td>
<td>DIANE</td>
<td>82/05/01</td>
<td>1983/05/01</td>
</tr>
<tr>
<td>MCCOY</td>
<td>JOHN</td>
<td>81/07/01</td>
<td>1982/07/01</td>
</tr>
<tr>
<td>MCKNIGHT</td>
<td>ROGER</td>
<td>82/02/02</td>
<td>1983/02/02</td>
</tr>
<tr>
<td>ROMANS</td>
<td>ANTHONY</td>
<td>82/07/01</td>
<td>1983/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>MARY</td>
<td>81/07/01</td>
<td>1982/07/01</td>
</tr>
<tr>
<td>SMITH</td>
<td>RICHARD</td>
<td>82/01/04</td>
<td>1983/01/04</td>
</tr>
<tr>
<td>STEVENS</td>
<td>ALFRED</td>
<td>80/06/02</td>
<td>1981/06/02</td>
</tr>
</tbody>
</table>
Index

-IF command 55
-RUN command 57, 58
-SET command 54, 55

A

ABS function 344
accessing external functions 64, 66
OS/390 64
UNIX 66
VM/CMS 66
accessing functions 43, 45, 46, 65
OS/390 65
ADD function 309, 310
adding function libraries 66
alphanumeric argument 47
alphanumeric strings 318
converting 318
ARGLEN function 68, 69
argument formats 47
argument length 48
argument types 46
ASCII character chart 34
ASIS function 69, 70, 345
assigning date-time values 271, 272, 273, 274
COMPUTE command 271, 272, 273
DEFINE command 271, 272
IF criteria 271, 272, 274
WHERE criteria 271, 272, 273
ATODBL function 318
AYM function 246, 247
AYMD function 247, 248

B

BAR function 345, 346, 347
batch allocation 64
bit strings 72, 73
BITSON function 71, 72
BITVAL function 72, 73
branching in procedures 55, 58
functions and 55, 58
BUSDAYS parameter 202
business days 202
BUSDAYS parameter 202
BYTVAL function 73, 74, 75

C

calling functions 43, 44, 45, 46, 52, 53, 58, 59, 60, 61, 62, 63
Dialogue Manager and 53
FOCUS commands and 52
from another function 58, 59
IF criteria 59, 60
in Maintain 46
RECAP command and 62, 63
WHEN criteria 61
WHERE criteria 59, 60
CHAR2INT function 152
character chart 34
ARGLEN 68, 69
ASIS 69, 70
BITSON 71, 72
BITVAL 72, 73
BYTVAL 73, 74, 75
CHKFMT 75, 77
CTRAN 78, 79, 80
CTRFLD 84, 85
DCTRAN 140
DSTRIP 142, 143
EDIT 85, 86, 87
GETTOK 87, 88, 89
LCWORD 89, 90, 91, 154
LCWORD2 91, 93
LCWORD3 92
LJUST 93
LOCASE 95
Maintain-specific 23, 151
OVRAY 96, 97
PARAG 99, 100
POSIT 104, 105, 106
RJUST 108, 109
SOUNDEX 109, 110
SPELLNM 111, 112
SQUEEZ 112, 113
STRIP 114, 115, 116
SUBSTR 118, 119, 120, 121, 133
TRIM 121, 122
TRIMV 135
UPCASE 123, 124
variable length 127
character strings (continued)
bits 71, 72
centering 84, 85
comparing 109, 169, 171
converting case 95, 123, 153, 156, 176
determining length 175
Dialogue Manager 69
dividing 99
extracting 156, 172
extracting characters 85
extracting substrings 87, 118, 121, 133, 173
finding substrings 104, 164
format 75
justifying 93, 108, 165
measuring length 68, 154
overlaying 96, 162
reducing spaces 112
removing occurrences 121, 175
right-justifying 108, 165
spelling out numbers 111
substrings 167, 171
translating characters 73, 78, 79, 152
CHGDAT function 249, 250, 251
CKFMT function 75, 77
CKPKCK function 347, 348
CLSDREC 380, 391
commands 389
passing 389
compiling subroutines 411
OS/390 411
components 268
COMPUTE command 52, 273
assigning date-time values 273
controlling function parameter verification 50
converting formats 318
creating subroutines 399, 400
cross-referenced data sources 191
CTRAN function 78, 79, 80
CTRFLD function 84, 85
custom subroutines 412, 413, 415, 418, 422, 425, 426, 427

D

DA functions 252
DADMY function 252
DADYM function 252
DAMDY function 252
DAMYD function 252
data sets 381, 383
data source functions 25, 177, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 196, 197
  FIND 185, 186, 187, 188
  LAST 189, 190
  LOOKUP 191, 192, 193, 194, 196, 197
data source values 177, 181, 185, 186, 187, 188, 189, 190
decoding 181
retrieving 189, 190
verifying 185, 186, 187, 188
data sources 177, 185, 186, 187, 188, 189, 190, 191, 196
cross-referenced 191, 196
retrieving values 189, 190, 191, 196
values 177
verifying values 185, 186, 187, 188
arguments and 268
date and time functions (continued)
  AYM 246, 247
  AYMD 247, 248
  CHGDAT 249, 250, 251
  DA 252
  DADMY 252
  DADYM 252
  DAMDY 252
  DAMYD 252
  DATEADD 207, 208, 209
  DATECVT 209, 210
  DATEDIF 211, 212, 213
  DATEMOV 214, 216, 217
  DATETRAN 217
  DAYDM 252
  DAYMD 252, 253
  DOWK 255
  DOWKL 255
  DTDMY 256
  DTMYD 256
  DTYDM 256
  GREGDT 258
  HADD 274, 275, 276
  HCNVRT 276, 277
  HDATE 278, 279
  HDIFF 279, 280
  HDTTM 281, 282
  HGETC 284, 285
  HHMSS 286
  HINPUT 287, 288
  HNAME 293, 294
  JULDAT 259, 260
  Maintain-specific 305
  standard 201
  TIMETOTS 299, 300
  TODAY 242
date and time functions (continued)

YM 261, 262
YMD 254
date argument 47
date formats 217, 269, 270
  formatted-string format 270
  international 217
  numeric string format 269
  translated-string format 270
date functions 25, 202
  work days 202
date-time format
  ISO standard input values 271
date-time functions 28, 282, 283, 290
  HETXR 282, 283
  HMASK 290
  adding 246, 247, 309
  assigning 271
  converting 298, 299
  converting formats 249, 252, 256, 259, 276, 278, 281
determining day of week 315
determining quarter 312
elapsed time 261, 311
extracting 310, 312, 316
finding day of week 255
finding difference 211, 253, 279
incrementing 274
moving dates 214
retrieving 306, 307, 308, 309
retrieving components 294
retrieving time 286
returning dates 242
setting 313
setting time 288
storing 284
subtracting 246, 247, 314

DATEADD function 207, 208, 209
DATECVT function 209, 210
DATEDIF function 211, 212, 213, 214
DATEFNS parameter 244, 258, 259
  GREGDT function 258
  JULDAT function 259
DATEFORMAT parameter 265
DATEMOV function 214, 216, 217
DATETRAN function 217, 224
DAY function 310, 311
DAYDM function 252
DAYMD function 252, 253
DB_LOOKUP function 178
  COMPUTE command 178
  DEFINE 178
  MODIFY 178
  TABLE COMPUTE 178
DCTRAN function 140
DECODE function 181, 182, 183, 184
decoding functions 25, 177, 181, 182, 183, 184
decoding values 181, 182, 183, 184
  from files 181, 183, 184
  in a function 181, 182, 183
DDEDIT function 141
DEFINE command 52
  functions and 52
deleting function libraries 66
Dialogue Manager 53
  functions and 53
Dialogue Manager commands 53, 54, 55, 57, 58
  -IF 55
  -RUN 57, 58
  -SET 54, 55
DMOD function 350, 351
DMY function 253, 254
double-byte characters 140, 142
DOWK function 255
DOWKL function 255
DPART function 232
DSTRIP function 142, 143
DSUBSTR function 143
DTDMY function 256
DTDYM function 256
DTMDY function 256, 257
DTMYD function 256
DTSTRICT parameter 267
DTYDM function 256
DTYMD function 256

F

FEXERR function 380, 381
FIND function 185, 186, 187, 188
FINDMEM function 381, 382, 383
FIQTR function 236
FIYR function 234
FIYYQ function 239
FML (Financial Modeling Language) 355, 357
retrieving tag lists 355
retrieving tag values 357
FML hierarchies 358
FMLCAP function 358
FMLFOR function 357
FMLINFO function 354, 355
FMLLIST function 355, 356
FMOD function 350, 351
FOCUS commands 52
FOR lists 355
retrieving 355
format conversion functions 30, 318, 321, 322,
328, 329, 330, 331, 332, 333, 334,
335, 336, 337, 338, 339, 340
ATOMDL 318
EDIT 321, 322
FPRINT 322
FTOA 328, 329
HEXBYT 330, 331
ITONUM 332, 333
ITOPACK 333, 334
ITOZ 335
PCKOUT 336, 337
PTOA 338, 339
UFMT 339, 340
format conversions 318, 328, 330, 332, 333,
335, 336, 338, 339
packed numbers 336
to alphanumeric 328, 338
to characters 330
to double-precision 332
to hexadecimal 339
to packed decimal 333

E

EBCDIC character chart 34
EDALIB.LOAD library 64
EDIT function 85, 86, 87, 321, 322
enabling parameter verification 49
entry points 406, 407
error messages 380
EXP function 351, 352
EXPN function 352
eXternal functions 18, 64

Using Functions
format conversions (continued)
  to zoned format 335
formats 318, 322
  alphanumeric 322
  converting 318
formatted-string format 270
FPRINT function 322
FTOA function 328, 329
function arguments 46, 47, 48, 58, 59, 401
  formats 47
  functions as 58, 59
  in subroutines 401
  length 48
  number 48
  types 46
function libraries 46, 66
  adding 66
  deleting 66
function types 19, 20, 22, 23, 25, 28, 30, 31, 33, 68, 151, 177, 343, 379, 391
  character 20, 22, 23, 68, 151
  data source 25, 177
  date 25
  date-time 28
  decoding 25, 177
  format conversion 30
  light update support 391
  Maintain-specific character 23
  numeric 31, 343
  system 33, 379
functions 18, 19, 43, 44, 45, 46, 52, 53, 54, 55, 57, 58, 59, 60, 66, 116, 127, 178, 185, 186, 187, 188, 234, 236, 239, 244, 282, 283, 290, 353, 354, 355, 356, 357, 358, 362, 374, 394, 399
  -IF command and 55
  -RUN command and 57, 58
  accessing 43
  branching in procedures 55, 60
  calling 43, 44, 45, 46, 52, 58, 59
  COMPUTE command and 52
  date and time 244
functions (continued)
  DEFINE command and 52
  Dialogue Manager and 53
  external 18
  FIND 185, 186, 187, 188
  FIQTR 236
  FIYR 234
  FIYYQ 239
  FMLCAP 358
  FMLFOR 357
  FMLINFO 353, 354, 355
  FMLLIST 355, 356
  FOCUS commands and 52
  HEXITR 282, 283
  HMASK 290
  internal 18
  invoking 45
  MIRR 362
  operating system commands and 57, 58
  SLEEP 394
  STRREP 116
  subroutines 18, 399
  types 19
  VALIDATE command and 52
  variable length character 127
  variables and 54, 55
  VM/CMS 66
  XIRR 374
functions, LCWORD3 92
FUSELIB.LOAD library 64

G

GETPDS function 383, 384, 385, 387
GETTOK function 87, 88, 89
GETUSER function 388
GREGDT function 258
  DATESFNS parameter 258
**H**

HADD function 274, 275, 276
HCNVRT function 276, 277
HDATE function 278, 279
HDAY parameter 203, 204
HDIFF function 279, 280
HDTTM function 281, 282
HEXBYTE function 330, 331
HEXTR function 282, 283
HGETC function 284, 285
HPART function 294, 295, 296
HMIDNT function 288, 289
HNAME function 293, 294
holidays 202, 203, 204
   HDAY parameter 204
   holiday files 203
HPART function 294, 295, 296
HSETPT function 296, 297
HTIME function 298, 299
HTMTOTS function 299
HYYWD function 301

**I**

IF criteria 59, 60, 274
   assigning date-time values 274
   functions and 59, 60
IMOD function 350, 351
Initial_HHMMSS function 307
Initial_TODAY 308
Initial_TODAY function 307
INT function 359
INT2CHAR function 152, 153
internal functions 18
   internal modified rate of return 362
   internal rate of return 374
   international date formats 217
   invoking functions 44, 45
   ISO standard date-time formats 271
   ITONUM function 332, 333
   ITOPACK function 333, 334
   ITOZ function 335

**J**

JULDAT function 259, 260
   DATEFNS parameter 259
JULIAN function 311

**L**

LAST function 189, 190
LCWORD function 89, 90, 91, 153, 154
LCWORD2 function 91, 93, 153, 154
LCWORD3 function 92
LEADZERO parameter 205
legacy date functions 25, 244, 245, 253, 254
   DATEFNS parameter 244
   DMY 253, 254
   legacy dates 245
   legacy versions 244
   MDY 253, 254
Index

legacy date functions (continued)
   YMD 253, 254
legacy dates 245
LENGTH function 154, 155
light update support functions 391
LJUST function 93, 156
load libraries 64
LOCAS function
   variable length 130
LOCASE function 95
LOG function 360
LOOKUP function 191, 192, 193, 194, 196, 197
   extended function 196, 197
LOWER function 156

M

Maintain data source functions 186, 187
   CHAR2INT 152
   INT2CHAR 152, 153
   LCWORD 153
   LCWORD2 91, 153, 154
   LENGTH 154, 155
   LJUST 156
   LOWER 156
   MASK 156, 157
   NLSCHR 162
   OVRLAY 162, 163
   POSIT 164, 165
   RJUST 165, 166
   SELECTS 166, 167
   STRAN 167, 168
   STRCMP 169, 170
   STRICMP 171
   STRTOKEN 172, 173

Maintain-specific character functions (continued)
   SUBSTR 173, 174
   TRIM 175
   TRIMLEN 175, 176
   UPCASE 176
Maintain-specific date and time functions 214, 217, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316
   ADD 309, 310
   DATEDIF 214
   DATEMOV 217
   DAY 310, 311
   HHMSS 306, 307
   Initial_HHMSS 307
   Initial_TODAY 307, 308
   JULIAN 311
   MONTH 312
   QUARTER 312, 313
   SETMDY 313, 314
   standard 305
   SUB 314, 315
   TODAY 308
   TODAY2 309
   WEEKDAY 315
   YEAR 316
Maintain-specific functions 46, 166, 171, 172, 391
   light update support 391
   MNTUWS function library 46
   SELECTS 166
   STRNCMP 171, 172
   STRTOKEN 172
   MASK function 156, 157
   MAX function 361
   MDY function 253, 254
   MIN function 361
   MIRR function 362
   MNTGETTOK function 158
   MNTUWS function library 46
   modified rate of return 362
MODIFY data source functions 187, 188
MONTH built-in function 312
MONTH function 312
MTHNAM subroutine 412, 413, 415, 418, 422, 425, 426, 427
MVSDYNAM function 389, 390

**N**
naming subroutines 401
National Language Support (NLS) 162
NLS (National Language Support) 162
NLSCHR function 162
NORMSDST function 365, 367, 368, 369
NORMSINV function 365, 367, 368, 369
number of arguments 48
numbers 344, 347, 350, 351, 359, 360, 361, 365, 367, 368, 370, 372, 373
  absolute value 344
  calculating remainders 350
  generating random 370, 372
  greatest integer 359
  logarithms 360
  maximum 361
  minimum 361
  raising to a power 351
  square root 373
  standard normal deviation 365, 367, 368
  validating packed fields 347
numeric argument 47
  ABS 344
  ASIS 345
numeric functions (continued)
  BAR 345, 346, 347
  CHKPCK 347, 348
  DMOD 350, 351
  EXP 351, 352
  FMLCAP 358
  FMLF0R 357
  FMLINFO 353, 354, 355, 356
  FMOD 350, 351
  IMOD 350, 351
  INT 359
  LOG 360
  MAX 361
  MIN 361
  NORMSDST 365, 367, 368, 369
  NORMSINV 365, 367, 368, 369
  PRDNOR 370, 371
  PRDUNI 370, 371
  RDNORM 372, 373
  RDUNIF 372, 373
  SQRT 373, 374
numeric string format 269
numeric values 343

**O**
operating system commands 57, 58
order of arguments 48
OS/390 64, 65, 66, 411
  compiling subroutines 411
  storing functions 64, 65, 66
  storing subroutines 411
OVRLAY function 96, 97, 162, 163

**P**
packed numbers, writing to an output file 341
PARAG function 99, 100
PATTERN function 101
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCKOUT function</td>
<td>336, 337</td>
</tr>
<tr>
<td>POSIT function</td>
<td>104, 105, 106, 164, 165</td>
</tr>
<tr>
<td>PRDNOR function</td>
<td>370, 371</td>
</tr>
<tr>
<td>PRDUNI function</td>
<td>370, 371</td>
</tr>
<tr>
<td>programming subroutines</td>
<td>405, 407, 409</td>
</tr>
<tr>
<td>arguments</td>
<td>407, 409</td>
</tr>
<tr>
<td>PTOA function</td>
<td>338, 339</td>
</tr>
<tr>
<td>PUTDDREC</td>
<td>391</td>
</tr>
</tbody>
</table>

**Q**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUARTER function</td>
<td>312, 313</td>
</tr>
</tbody>
</table>

**R**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate of return</td>
<td>362, 374</td>
</tr>
<tr>
<td>RDNORM function</td>
<td>372, 373</td>
</tr>
<tr>
<td>RDUNIF function</td>
<td>372, 373</td>
</tr>
<tr>
<td>RECAP command</td>
<td>62, 63</td>
</tr>
<tr>
<td>retrieving FML hierarchy captions</td>
<td>358</td>
</tr>
<tr>
<td>return rate functions</td>
<td>362, 374</td>
</tr>
<tr>
<td>MIRR 362</td>
<td></td>
</tr>
<tr>
<td>XIRR 374</td>
<td></td>
</tr>
<tr>
<td>REVERSE function</td>
<td>106</td>
</tr>
<tr>
<td>REXX subroutines</td>
<td>427, 429, 430, 431, 432, 433, 434, 435, 436, 437</td>
</tr>
<tr>
<td>formats 433</td>
<td></td>
</tr>
<tr>
<td>RJUST function</td>
<td>108, 109, 165, 166</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTS function</td>
<td>166, 167</td>
</tr>
<tr>
<td>SET parameters</td>
<td>202, 203, 204, 205, 244, 267</td>
</tr>
<tr>
<td>BUSDAYS 202</td>
<td></td>
</tr>
<tr>
<td>DATEFNS 244</td>
<td></td>
</tr>
<tr>
<td>SET parameters (continued)</td>
<td></td>
</tr>
<tr>
<td>DTSTRICT 267</td>
<td></td>
</tr>
<tr>
<td>HDAY 203, 204</td>
<td></td>
</tr>
<tr>
<td>LEADZERO 205</td>
<td></td>
</tr>
<tr>
<td>SETMDY function</td>
<td>313, 314</td>
</tr>
<tr>
<td>single-byte characters</td>
<td>140, 142</td>
</tr>
<tr>
<td>SLEEP function</td>
<td>394</td>
</tr>
<tr>
<td>SOUNDEX function</td>
<td>109, 110</td>
</tr>
<tr>
<td>SPELLNM function</td>
<td>111, 112</td>
</tr>
<tr>
<td>SQRT function</td>
<td>373, 374</td>
</tr>
<tr>
<td>SQUEEZ function</td>
<td>112, 113</td>
</tr>
<tr>
<td>standard date and time functions</td>
<td>201</td>
</tr>
<tr>
<td>standard date functions</td>
<td>25</td>
</tr>
<tr>
<td>standard normal deviation</td>
<td>365, 367, 368</td>
</tr>
<tr>
<td>storing external functions</td>
<td></td>
</tr>
<tr>
<td>OS/390 64, 65, 66</td>
<td></td>
</tr>
<tr>
<td>UNIX 66</td>
<td></td>
</tr>
<tr>
<td>VM/CMS 66</td>
<td></td>
</tr>
<tr>
<td>storing subroutines</td>
<td>411</td>
</tr>
<tr>
<td>OS/390 411</td>
<td></td>
</tr>
<tr>
<td>STRAN function</td>
<td>167, 168</td>
</tr>
<tr>
<td>STRCMP function</td>
<td>169, 170</td>
</tr>
<tr>
<td>STRICMP function</td>
<td>171</td>
</tr>
<tr>
<td>string replacement</td>
<td>116</td>
</tr>
<tr>
<td>STRIP function</td>
<td>114, 115, 116</td>
</tr>
<tr>
<td>STRNCMP function</td>
<td>171, 172</td>
</tr>
<tr>
<td>STRREP function</td>
<td>116</td>
</tr>
<tr>
<td>STRTOKEN function</td>
<td>172, 173</td>
</tr>
<tr>
<td>SUB function</td>
<td>314, 315</td>
</tr>
<tr>
<td>compiling 411</td>
<td></td>
</tr>
</tbody>
</table>
subroutines (continued)
creating 399, 400
custom 412, 413, 415, 418, 422, 425, 426, 427
ten points 406, 407
MTHNAM 412, 413, 415, 418, 422, 425, 426, 427
naming 401
programming 405
REXX 427, 429, 430, 431, 432, 433, 434, 435, 436, 437
storing 411
testing 411
writing 400
SUBSTR function 118, 119, 120, 121, 133, 173, 174
variable length 133
substrings 85, 87, 96, 104, 118, 121, 133, 162, 164, 167, 171, 173
comparing 171
extracting 85, 87, 118, 121, 133, 173
finding 104, 164
overlaying character strings 96, 162
substituting 167
system functions 33, 379, 380, 381, 382, 383, 384, 385, 387, 389, 390, 396
FEXERR 380, 381
FINDMEM 381, 382, 383
GETPDS 383, 384, 385, 387
GETUSER 388
MVSDYNAM 389, 390
SYSVAR 396
SYVAR function 396

T

tag lists 355
retrieving 355
tag values 357
testing subroutines 411
time formats 270, 271

TIMETOTS function 299, 300
TODAY function 242, 308
TODAY2 function 309
translated-string format 270
TRIM function 121, 122, 175
TRIMLEN function 175, 176
TRIMV function 135

U

UFMT function 339, 340
UNIX 66
  accessing functions 66
  storing functions 66
UPCASE function 123, 124, 176
user IDs 388
USERFCHK setting 48, 50
USERFNS setting 49

V

VALIDATE command 52
values 181, 185, 186, 187, 188
  decoding 181
  verifying 185, 186, 187, 188
variable length character functions 127
verifying function parameters 48, 49, 50
  controlling 50
  enabling 49
VM/CMS 66
  accessing external functions 66
  storing external functions 66
W

WEEKDAY function 315

WEEKFIRST parameter 265

WHEN criteria 61

WHERE criteria 59, 60, 273
  assigning date-time values 273
  functions and 59, 60
work days 202, 203
  business days 202
  holidays 202, 203
writing subroutines 400, 401, 402, 405, 406, 407, 409
  creating arguments 401
  entry points 406, 407
  languages 402
writing subroutines (continued)
  naming subroutines 401
  programming 405, 407, 409

X

XIRR function 374

XTPACK function 341

Y

YEAR function 316

YM function 261, 262

YMD function 253, 254
Reader Comments

In an ongoing effort to produce effective documentation, the Technical Content Management staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please share your suggestions for improving this publication and alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

**Mail:**
Technical Content Management
Information Builders, Inc.
Two Penn Plaza
New York, NY 10121-2898

**Fax:**
(212) 967-0460

**Email:**
books_info@ibi.com

**Website**
http://documentation.informationbuilders.com/connections.asp

Name:__________________________________________________________
Company:_____________________________________________________
Address:_______________________________________________________
Telephone:_________________________ Date:_______________________
Email:_________________________________________________________
Comments:
FOCUS for Mainframe

Using Functions

Version 7.7