# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>13</td>
</tr>
<tr>
<td>Documentation Conventions</td>
<td>13</td>
</tr>
<tr>
<td>Related Publications</td>
<td>14</td>
</tr>
<tr>
<td>Customer Support</td>
<td>14</td>
</tr>
<tr>
<td>Information You Should Have</td>
<td>15</td>
</tr>
<tr>
<td>User Feedback</td>
<td>16</td>
</tr>
<tr>
<td>Information Builders Consulting and Training</td>
<td>16</td>
</tr>
<tr>
<td><strong>1. Language Rules Reference</strong></td>
<td>17</td>
</tr>
<tr>
<td>Case Sensitivity</td>
<td>18</td>
</tr>
<tr>
<td>Specifying Names</td>
<td>19</td>
</tr>
<tr>
<td>Reserved Words</td>
<td>21</td>
</tr>
<tr>
<td>What Can You Include in a Procedure?</td>
<td>22</td>
</tr>
<tr>
<td>Multi-Line Commands</td>
<td>23</td>
</tr>
<tr>
<td>Terminating Command Syntax</td>
<td>24</td>
</tr>
<tr>
<td>Adding Comments</td>
<td>24</td>
</tr>
<tr>
<td><strong>2. Expressions Reference</strong></td>
<td>27</td>
</tr>
<tr>
<td>Types of Expressions You Can Write</td>
<td>27</td>
</tr>
<tr>
<td>Reference: Usage Notes for Expressions</td>
<td>28</td>
</tr>
<tr>
<td>Expressions and Variable Formats</td>
<td>28</td>
</tr>
<tr>
<td>Writing Numeric Expressions</td>
<td>29</td>
</tr>
<tr>
<td>Reference: Numeric Operators</td>
<td>29</td>
</tr>
<tr>
<td>Syntax: How to Use DIV: Integer Division</td>
<td>30</td>
</tr>
<tr>
<td>Syntax: How to Use MOD: Calculating the Remainder</td>
<td>30</td>
</tr>
<tr>
<td>Order of Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>Evaluating Numeric Expressions</td>
<td>31</td>
</tr>
<tr>
<td>Identical Operand Formats</td>
<td>31</td>
</tr>
<tr>
<td>Different Operand Formats</td>
<td>32</td>
</tr>
<tr>
<td>Continental Decimal Notation</td>
<td>33</td>
</tr>
<tr>
<td>Writing Date Expressions</td>
<td>34</td>
</tr>
<tr>
<td>Formats for Date Values</td>
<td>34</td>
</tr>
<tr>
<td>Evaluating Date Expressions</td>
<td>35</td>
</tr>
<tr>
<td>Selecting the Format of the Result Variable</td>
<td>36</td>
</tr>
<tr>
<td>Manipulating Dates in Date Format</td>
<td>36</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Using a Date Constant in an Expression</td>
<td>37</td>
</tr>
<tr>
<td>Extracting a Date Component</td>
<td>37</td>
</tr>
<tr>
<td>Combining Variables With Different Components in an Expression</td>
<td>37</td>
</tr>
<tr>
<td>Different Operand Date Formats</td>
<td>38</td>
</tr>
<tr>
<td>Using Addition and Subtraction in a Date Expression</td>
<td>39</td>
</tr>
<tr>
<td>Writing Date-Time Expressions</td>
<td>40</td>
</tr>
<tr>
<td>Syntax: How to Write Date-Time Expressions</td>
<td>40</td>
</tr>
<tr>
<td>Manipulating Date-Time Values Directly</td>
<td>42</td>
</tr>
<tr>
<td>Comparing and Assigning Date-Time Values</td>
<td>42</td>
</tr>
<tr>
<td>Date-Time Subroutines</td>
<td>43</td>
</tr>
<tr>
<td>Reference: Notes Regarding ISO Standard Date-Time Representations</td>
<td>44</td>
</tr>
<tr>
<td>Writing Character Expressions</td>
<td>45</td>
</tr>
<tr>
<td>Concatenating Character Strings</td>
<td>46</td>
</tr>
<tr>
<td>Evaluating Character Expressions</td>
<td>46</td>
</tr>
<tr>
<td>Variable-Length Character Variables</td>
<td>48</td>
</tr>
<tr>
<td>Writing Logical Expressions</td>
<td>49</td>
</tr>
<tr>
<td>Relational Expressions</td>
<td>50</td>
</tr>
<tr>
<td>Boolean Expressions</td>
<td>50</td>
</tr>
<tr>
<td>Syntax: How to Use Boolean Expressions</td>
<td>50</td>
</tr>
<tr>
<td>Evaluating Logical Expressions</td>
<td>50</td>
</tr>
<tr>
<td>Reference: Logical Operators</td>
<td>51</td>
</tr>
<tr>
<td>Writing Conditional Expressions</td>
<td>52</td>
</tr>
<tr>
<td>Syntax: How to Use Conditional Expressions</td>
<td>52</td>
</tr>
<tr>
<td>Handling Null Values in Expressions</td>
<td>53</td>
</tr>
<tr>
<td>Assigning Null Values: The MISSING Constant</td>
<td>53</td>
</tr>
<tr>
<td>Syntax: How to Assign Null Values: The MISSING Constant</td>
<td>53</td>
</tr>
<tr>
<td>Conversion in Mixed-Format Null Expressions</td>
<td>54</td>
</tr>
<tr>
<td>Testing Null Values</td>
<td>54</td>
</tr>
<tr>
<td>Syntax: How to Test Null Values</td>
<td>54</td>
</tr>
<tr>
<td>3. Command Reference</td>
<td>55</td>
</tr>
<tr>
<td>Language Summary</td>
<td>55</td>
</tr>
<tr>
<td>Defining a Procedure</td>
<td>55</td>
</tr>
<tr>
<td>Defining a Maintain Function (a Case)</td>
<td>55</td>
</tr>
<tr>
<td>Defining Blocks of Code</td>
<td>56</td>
</tr>
</tbody>
</table>
Transferring Control. ................................................................. 56
Executing Procedures. ............................................................ 56
Encrypting Files. ................................................................. 57
Using Loops. ............................................................................. 57
Using Forms. ............................................................................. 57
Defining Classes. ................................................................. 57
Creating Variables. ............................................................... 58
Assigning Values. ............................................................... 58
Manipulating Stacks. ............................................................. 58
Selecting and Reading Records. ............................................ 59
Conditional Actions. ............................................................. 60
Writing Transactions. ............................................................ 60
Setting WebFOCUS Server Parameters. ................................ 61
Configuring a Shared Application Server. ............................. 62
Using Libraries of Classes and Functions. .............................. 62
Messages and Logs. ............................................................... 62
BEGIN .................................................................................. 63
   Syntax: How to Use the BEGIN Command. ............................ 63
CALL .................................................................................. 64
   Syntax: How to Use the CALL Command. ............................... 65
CASE .................................................................................. 69
   Syntax: How to Use the CASE Command. ............................ 70
   Reference: Usage Notes for CASE. ...................................... 71
   Reference: Commands Related to CASE. ............................... 71
   Calling a Function: Flow of Control. .................................... 71
   Passing Parameters to a Function. ........................................ 71
   Using the Return Value of a Function. .................................. 73
   Using the Top Function. ...................................................... 73
COMMIT ............................................................................. 73
   Syntax: How to Use the COMMIT Command. ........................ 74
   Reference: Usage Notes for COMMIT. ................................... 74
COMPUTE ........................................................................... 74
   Syntax: How to Use the COMPUTE Command. ...................... 75
   Reference: Usage Notes for COMPUTE. ............................... 78
   Reference: Commands Related to COMPUTE. ....................... 78
Using COMPUTE to Call Functions ......................................................... 79
COPY .................................................................................................................. 79
Syntax: How to Use the COPY Command ....................................................... 79
Reference: Usage Notes for COPY ................................................................. 82
Reference: Commands Related to COPY ....................................................... 83
DECLARE .......................................................................................................... 84
Syntax: How to Use the DECLARE Command ................................................ 84
Reference: Commands Related to DECLARE ............................................... 86
Local and Global Declarations ................................................................. 86
DECRIPT ............................................................................................................. 87
DELETE ............................................................................................................. 87
Syntax: How to Use the DELETE Command ................................................. 87
Reference: Usage Notes for DELETE ............................................................. 89
Reference: Commands Related to DELETE ................................................. 90
DESCRIBE .......................................................................................................... 90
Syntax: How to Use the DESCRIBE Command ........................................... 91
Class Member Functions ............................................................................. 93
Reference: Startup Case Considerations .................................................... 94
Reference: Executing Member Functions .................................................... 94
Defining and Using SuperClasses and Subclasses ..................................... 100
Reference: Commands Related to DESCRIBE ......................................... 103
ENCRYPT/DECRYPT ...................................................................................... 104
Encrypting Procedures ................................................................................ 104
Encrypting Data ........................................................................................... 104
Performance Considerations ....................................................................... 105
Restricting Existing Data Sources .............................................................. 106
END ..................................................................................................................... 106
Syntax: How to Use the END Command ...................................................... 107
Reference: Commands Related to END ....................................................... 107
EXEC .................................................................................................................. 107
Syntax: How to Use the EXEC Command .................................................... 107
FocCount ......................................................................................................... 109
FocCurrent ...................................................................................................... 109
FocError .......................................................................................................... 109
FocErrorRow .................................................................................................. 110
## Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FocFetch</td>
<td>110</td>
</tr>
<tr>
<td>FocIndex</td>
<td>110</td>
</tr>
<tr>
<td>FocMsg</td>
<td>111</td>
</tr>
<tr>
<td>GOTO</td>
<td>112</td>
</tr>
<tr>
<td>Syntax: How to Use the GOTO Command</td>
<td>112</td>
</tr>
<tr>
<td>Reference: Usage Notes for GOTO</td>
<td>114</td>
</tr>
<tr>
<td>Reference: Commands Related to GOTO</td>
<td>114</td>
</tr>
<tr>
<td>Using GOTO With Data Source Commands</td>
<td>114</td>
</tr>
<tr>
<td>GOTO and ENDCASE</td>
<td>114</td>
</tr>
<tr>
<td>GOTO and PERFORM</td>
<td>114</td>
</tr>
<tr>
<td>IF</td>
<td>115</td>
</tr>
<tr>
<td>Syntax: How to Use the IF Command</td>
<td>115</td>
</tr>
<tr>
<td>Coding Conditional COMPUTE Commands</td>
<td>118</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>118</td>
</tr>
<tr>
<td>Syntax: How to Use the INCLUDE Command</td>
<td>118</td>
</tr>
<tr>
<td>Reference: Usage Notes for INCLUDE</td>
<td>120</td>
</tr>
<tr>
<td>Reference: Commands Related to INCLUDE</td>
<td>121</td>
</tr>
<tr>
<td>Data Source Position</td>
<td>121</td>
</tr>
<tr>
<td>Null Values</td>
<td>121</td>
</tr>
<tr>
<td>INFER</td>
<td>121</td>
</tr>
<tr>
<td>Syntax: How to Use the INFER Command</td>
<td>122</td>
</tr>
<tr>
<td>Reference: Commands Related to INFER</td>
<td>123</td>
</tr>
<tr>
<td>Defining Non-Data Source Columns</td>
<td>123</td>
</tr>
<tr>
<td>MAINTAIN</td>
<td>123</td>
</tr>
<tr>
<td>Syntax: How to Use the MAINTAIN Command</td>
<td>124</td>
</tr>
<tr>
<td>Reference: Usage Notes for MAINTAIN</td>
<td>125</td>
</tr>
<tr>
<td>Reference: Commands Related to MAINTAIN</td>
<td>125</td>
</tr>
<tr>
<td>Specifying Data Sources With the MAINTAIN Command</td>
<td>125</td>
</tr>
<tr>
<td>Calling a Procedure From Another Procedure</td>
<td>125</td>
</tr>
<tr>
<td>MATCH</td>
<td>126</td>
</tr>
<tr>
<td>Syntax: How to Use the MATCH Command</td>
<td>127</td>
</tr>
<tr>
<td>Reference: Commands Related to MATCH</td>
<td>128</td>
</tr>
<tr>
<td>How the MATCH Command Works</td>
<td>128</td>
</tr>
<tr>
<td>MNTCON CDN_FEXINPUT</td>
<td>129</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON CDN_FEXINPUT Command</td>
<td>129</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNTCON COMPILE</td>
<td>130</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON COMPILE Command.</td>
<td>130</td>
</tr>
<tr>
<td>Reference: Commands Related to MNTCON COMPILE.</td>
<td>131</td>
</tr>
<tr>
<td>MNTCON EX</td>
<td>131</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON EX Command.</td>
<td>131</td>
</tr>
<tr>
<td>Reference: Commands Related to MNTCON EX.</td>
<td>132</td>
</tr>
<tr>
<td>Invoking Maintain Procedures: Passing Parameters.</td>
<td>132</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON EX Command to Pass Parameters.</td>
<td>132</td>
</tr>
<tr>
<td>MNTCON EXIT_WARNING</td>
<td>133</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON EXIT_WARNING Command.</td>
<td>134</td>
</tr>
<tr>
<td>MNTCON LOADIMAGE</td>
<td>134</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON LOADIMAGE Command.</td>
<td>134</td>
</tr>
<tr>
<td>Reference: Commands Related to MNTCON LOADIMAGE.</td>
<td>135</td>
</tr>
<tr>
<td>MNTCON MATCH_CASE</td>
<td>135</td>
</tr>
<tr>
<td>Syntax: How to Enable Mixed-Case Naming.</td>
<td>135</td>
</tr>
<tr>
<td>MNTCON PERFORMANCE_ANALYSIS</td>
<td>136</td>
</tr>
<tr>
<td>Syntax: How to Enable Maintain Performance Analysis on the Server.</td>
<td>136</td>
</tr>
<tr>
<td>MNTCON RADIO_BUTTON_EMIT_TEXT</td>
<td>137</td>
</tr>
<tr>
<td>Syntax: How to Send FOCINDEX From a Radio Button Web Link Event.</td>
<td>137</td>
</tr>
<tr>
<td>MNTCON REMOTESTYLE</td>
<td>137</td>
</tr>
<tr>
<td>Syntax: How to Disable the Variable Server Name Feature.</td>
<td>137</td>
</tr>
<tr>
<td>MNTCON RUN</td>
<td>138</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON RUN Command.</td>
<td>138</td>
</tr>
<tr>
<td>Reference: Commands Related to MNTCON RUN.</td>
<td>139</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON RUN Command to Pass Parameters.</td>
<td>139</td>
</tr>
<tr>
<td>MNTCON RUNIMAGE</td>
<td>140</td>
</tr>
<tr>
<td>Syntax: How to Use the MNTCON RUNIMAGE Command.</td>
<td>140</td>
</tr>
<tr>
<td>Reference: Commands Related to MNTCON RUNIMAGE.</td>
<td>140</td>
</tr>
<tr>
<td>MODULE</td>
<td>141</td>
</tr>
<tr>
<td>Syntax: How to Use the MODULE Command.</td>
<td>141</td>
</tr>
<tr>
<td>Reference: Commands Related to MODULE.</td>
<td>141</td>
</tr>
<tr>
<td>What You Can and Cannot Include in a Library.</td>
<td>141</td>
</tr>
<tr>
<td>NEXT</td>
<td>142</td>
</tr>
<tr>
<td>Syntax: How to Use the NEXT Command.</td>
<td>142</td>
</tr>
<tr>
<td>Reference: Usage Notes for NEXT.</td>
<td>144</td>
</tr>
</tbody>
</table>
This documentation provides a reference for the Maintain language. It is intended for application developers who are responsible for planning the enterprise software environment and for operating WebFOCUS Maintain.

How This Manual Is Organized

This manual includes the following chapters:

<table>
<thead>
<tr>
<th>Chapter/Appendix</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Language Rules Reference</td>
<td>Describes the rules for using the Maintain language.</td>
</tr>
<tr>
<td>2 Expressions Reference</td>
<td>Describes the rules for combining variables, constants, operators, and functions to produce different types of expressions.</td>
</tr>
<tr>
<td>3 Command Reference</td>
<td>Describes the purpose, syntax, and usage notes of all Maintain commands and system variables.</td>
</tr>
<tr>
<td>A WebFOCUS Maintain Error Messages</td>
<td>Lists all of the WebFOCUS Maintain error messages.</td>
</tr>
</tbody>
</table>

Documentation Conventions

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

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THIS TYPEFACE or this typeface</td>
<td>Denotes syntax that you must enter exactly as shown.</td>
</tr>
<tr>
<td>this typeface</td>
<td>Represents a placeholder (or variable) in syntax for a value that you or the system must supply.</td>
</tr>
<tr>
<td>underscore</td>
<td>Indicates a default setting.</td>
</tr>
</tbody>
</table>
### Convention | Description
--- | ---
*this typeface* | Represents a placeholder (or variable), a cross-reference, or an important term. It may also indicate a button, menu item, or dialog box option that you can click or select.

| Key + Key | Indicates keys that you must press simultaneously. |
| { } | Indicates two or three choices. Type one of them, not the braces. |
| [ ] | Indicates a group of optional parameters. None are required, but you may select one of them. Type only the parameter in the brackets, not the brackets. |
| | Separates mutually exclusive choices in syntax. Type one of them, not the symbol. |
| ... | Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis (...). |
| . . . | Indicates that there are (or could be) intervening or additional commands. |

### Related Publications
Visit our Technical Documentation Library at [http://documentation.informationbuilders.com](http://documentation.informationbuilders.com). You can also contact the Publications Order Department at (800) 969-4636.

### Customer Support
Do you have any questions about this product?

Join the Focal Point community. Focal Point is our online developer center and more than a message board. It is an interactive network of more than 3,000 developers from almost every profession and industry, collaborating on solutions and sharing tips and techniques. Access Focal Point at [http://forums.informationbuilders.com/eve/forums](http://forums.informationbuilders.com/eve/forums).
You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our website, http://www.informationbuilders.com. 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).
- Your WebFOCUS configuration:
  - The front-end software you are using, including vendor and release.
  - The communications protocol (for example, TCP/IP or HLLAPI), including vendor and release.
  - The software release.
  - Your server version and release. You can find this information using the Version option in the Web Console.
  - The stored procedure (preferably with line numbers) or SQL statements being used in server access.
- The Master File and Access File.
- The exact nature of the problem:
  - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
  - Provide the error message and return code, if applicable.
Is this related to any other problem?

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

What release of the operating system are you using? Has it, your security system, communications protocol, or front-end software changed?

Is this problem reproducible? If so, how?

Have you tried to reproduce your problem in the simplest form possible? For example, if you are having problems joining two data sources, have you tried executing a query containing just the code to access the data source?

Do you have a trace file?

How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

User Feedback

In an effort to produce effective documentation, the Technical Content Management staff welcomes your opinions regarding this document. 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.
You can use the Maintain language more effectively if you are familiar with its standards, including:

- When to use uppercase and lowercase characters.
- When to spell out keywords in full.
- How to name fields, functions, and other procedure components.
- Which words to avoid using as names of procedure components.
- What sort of components you can include in a procedure.
- How to continue a command onto additional lines.
- How to terminate command syntax.
- How to include comments in a procedure.

Data source descriptions and WebFOCUS procedures are not part of the Maintain language and are subject to different rules. The language rules for data source descriptions are discussed in *Describing Data With WebFOCUS Language*. The language rules for WebFOCUS procedures are discussed in *Creating Reports With WebFOCUS Language*.

**In this chapter:**

- Case Sensitivity
- Specifying Names
- Reserved Words
- What Can You Include in a Procedure?
- Multi-Line Commands
- Terminating Command Syntax
- Adding Comments
Case Sensitivity

By default, Maintain does not usually distinguish between uppercase and lowercase letters. You can enter keywords and names (such as data source and field names) in any combination of uppercase and lowercase letters. The only two exceptions are the MAINTAIN and END keywords used to begin and end a request. These keywords must be uppercase.

However, if mixed-case or NLS field and segment names are used in your application by enabling the MNTCON MATCH_CASE command, you must be consistent with the case style used in the names you give your variables and other application components. In addition, function names must match exactly as written in the documentation. However, Maintain keywords such as Repeat and Include do not need any special consideration when using this feature. For a full discussion of MNTCON MATCH_CASE, see MNTCON MATCH_CASE on page 135.

For example, the following ways of specifying the REPEAT command are equally valid, and Maintain always considers them to be identical:

REPEAT
repeat
RePeat
REPeat

By default, you can mix uppercase and lowercase to make variable names more understandable to a reader. For example, the stack name SALARYSTACK could also be represented as SalaryStack.

You may notice that when this manual presents sample Maintain source code, it shows keywords in uppercase, and user-defined names (such as field and stack names) in mixed case. This is only a documentation convention, not a Maintain language rule. As already explained, you can code Maintain commands in uppercase and lowercase.

While, by default, Maintain is not sensitive to the case of syntax, it is sensitive to the case of data. For example, the MATCH command distinguishes between the values SMITH and Smith.

Note: While you do not need to worry about case in Maintain procedures, any Master Files that Maintain accesses must have field and segment names in uppercase unless the MATCH_CASE feature is enabled.
Specifying Names

Maintain offers you a great deal of flexibility when naming and referring to procedure components, such as fields, functions, Winform buttons, and stacks. When naming a component, be aware of the following guidelines:

- **Length of names.** Unqualified names that are defined in a Maintain procedure (such as the unqualified names of Winforms, functions, and stacks) can be up to 66 characters long.

  There is no limit on the length of a qualified name, as long as the length of each of its component unqualified names does not exceed 66 characters.

  Master File names, and names defined within a Master File (such as names of fields and segments), are subject to standard Master File language conventions, as defined in *Describing Data With WebFOCUS Language*.

  Procedure name length is dependent on the operating system. On the Mainframe, procedure names can be a maximum of eight characters long.

- **Valid characters in a name.** All names must begin with a letter, and can include any combination of letters, numbers, and underscores (_).

  The names of projects and deployment scenarios can also include embedded spaces. (Other types of names cannot include spaces.)

- **Identical names.** Most types of items in a Maintain procedure can have the same name. The only exceptions are data sources, stacks, and Winforms, which cannot have the same name within the same Maintain procedure. Most types of items in a Maintain procedure can have the same name, but this is not recommended.

  For example, you may give the same name to fields in different segments, data sources, and stacks, and to controls in different Winforms, as long as you prevent ambiguous references by qualifying the names. A data source, a stack, and a Winform used in the same procedure can never have the same name.

- **Qualified names.** In general, whenever you can qualify a name, you should do so.

  Maintain requires that the qualification character be a period (.). The QUALCHAR parameter of the SET command must therefore be set (to the default).
If a qualified name cannot fit onto the current line, you can break the name at the end of any one of its components, and continue it onto the next line. The continued name must begin with the qualification character. In the following example, the continued line is indented for clarity:

```
FOR ALL NEXT ThisIsAVeryLongDataSourceName.ThisIsAVeryLongSegmentName .ThisIsAVeryLongFieldName INTO CreditStack;
```

You can qualify the names of:

- **Controls.** You can qualify a control name with the name of the Winform in which it is found. For example, if a button named UpdateButton is in a form named CreditForm, you could refer to the button as:

  `CreditForm.UpdateButton`

- **Member functions and member variables.** When referring to the member functions and member variables of an object, you should always use the fully-qualified name of the function or variable (that is, the name in the Winform `objectname.functionname` or `objectname.variablename`).

- **Fields and columns.** You can qualify a variable name with the name of the data source, segment, and/or stack in which it is found, using a period (.) as the qualification character.

Qualification is important when:

- You are working with two data sources in one Maintain procedure, and the data sources have field names in common.

- A field is present in both a data source and a stack, but it is not clear from the context which one is being referred to.

- Different Winforms in the same procedure include identically-named controls.

For example, both the Employee and JobFile data sources have a field named JobCode. If you want to issue a NEXT command for the JobCode field in Employee, you would use a qualified field name:

```
NEXT Employee.JobCode;
```

You can qualify a field name with any combination of its data source, segment, and stack names. When including a stack name, you have the option of specifying a particular row in the stack. If you use several qualifiers, they must conform to the following order:

```
stackname(row).datasourcename.segmentname.fieldname
```
If you refer to a field using a single qualifier (such as Sales) in the example

```
Sales.Quantity
```

and the qualifier is the name of both a segment and a stack, Maintain assumes that the name refers to the stack. To refer to the segment in this case, use the data source qualifier.

- **Truncated names.** You must spell all names in full. Maintain does not recognize truncated names, such as Dep for a field named Department.

- **Name aliases.** You cannot refer to a field by its alias in a Maintain procedure. (An alias is defined by the ALIAS attribute of a field in a Master File.)

## Reserved Words

The words in the following table are Maintain keywords and are reserved. You may not use them as identifiers. Identifiers are names of project components (such as, but not limited to, classes, functions, data sources, data source segments, stacks, stack columns, scalar variables, and Winforms).

In addition to these words, you may not use the names of built-in functions to name functions that you create yourself. See the Using Functions manual for a complete list of built-in functions.

If a procedure uses an existing Master File that employs a reserved word as a field name, you can refer to the field by qualifying its name with the name of the segment or data source.

**Note:** All reserved words can be written in any case, even when using MNTCON MATCH_CASE ON.

<table>
<thead>
<tr>
<th>ALL</th>
<th>AND</th>
<th>AS</th>
<th>ASK</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BIND</td>
<td>BY</td>
<td>CALL</td>
<td>CASE</td>
</tr>
<tr>
<td>CFUN</td>
<td>CLASS</td>
<td>CLEAR</td>
<td>COMMIT</td>
<td>COMPUTE</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>CONTENTS</td>
<td>COPY</td>
<td>current</td>
<td>DATA</td>
</tr>
<tr>
<td>DECLARE</td>
<td>DECODE</td>
<td>DELETE</td>
<td>DEPENDENTS</td>
<td>DESCRIBE</td>
</tr>
<tr>
<td>DFC</td>
<td>DIV</td>
<td>DROP</td>
<td>DUMP</td>
<td>ELSE</td>
</tr>
<tr>
<td>END</td>
<td>ENDBEGIN</td>
<td>ENDCASE</td>
<td>ENDDESCRIBE</td>
<td>ENDREPEAT</td>
</tr>
<tr>
<td>EQ</td>
<td>EQ_MASK</td>
<td>ERRORS</td>
<td>EVENT</td>
<td>EXCEEDS</td>
</tr>
<tr>
<td>EXEC</td>
<td>EXECSQL</td>
<td>EXIT</td>
<td>EXITREPEAT</td>
<td>EXPORT</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>FALSE</td>
<td>FILE</td>
<td>FILES</td>
<td>FIND</td>
<td>FocCount</td>
</tr>
<tr>
<td>FocCurrent</td>
<td>FocEnd</td>
<td>FocEndCase</td>
<td>FocEOF</td>
<td>FocError</td>
</tr>
<tr>
<td>FocErrorRow</td>
<td>FocIndex</td>
<td>FOR</td>
<td>FROM</td>
<td>GE</td>
</tr>
<tr>
<td>GET</td>
<td>GOTO</td>
<td>GT</td>
<td>HERE</td>
<td>HIGHEST</td>
</tr>
<tr>
<td>HOLD</td>
<td>IF</td>
<td>IN</td>
<td>IMPORT</td>
<td>INCLUDE</td>
</tr>
<tr>
<td>INFER</td>
<td>INTO</td>
<td>IS</td>
<td>IS_LESS_THAN</td>
<td>IS_NOT</td>
</tr>
<tr>
<td>KEEP</td>
<td>LE</td>
<td>LIKE</td>
<td>LT</td>
<td>MAINTAIN</td>
</tr>
<tr>
<td>MATCH</td>
<td>MISSING</td>
<td>MOD</td>
<td>MODULE</td>
<td>MOVE</td>
</tr>
<tr>
<td>NE</td>
<td>NE_MASK</td>
<td>NEEDS</td>
<td>NEXT</td>
<td>NO</td>
</tr>
<tr>
<td>NOT</td>
<td>NOWAIT</td>
<td>OBJECT</td>
<td>OF</td>
<td>OFF</td>
</tr>
<tr>
<td>OMITS</td>
<td>ON</td>
<td>OR</td>
<td>PATH</td>
<td>PERFORM</td>
</tr>
<tr>
<td>QUIT</td>
<td>REPEAT</td>
<td>REPOSITION</td>
<td>RESET</td>
<td>RETURN</td>
</tr>
<tr>
<td>RETURNS</td>
<td>REVISE</td>
<td>ROLLBACK</td>
<td>SAY</td>
<td>SELECTS</td>
</tr>
<tr>
<td>self</td>
<td>SET</td>
<td>SOME</td>
<td>SORT</td>
<td>SQL</td>
</tr>
<tr>
<td>STACK</td>
<td>TAKES</td>
<td>THEN</td>
<td>TO</td>
<td>TOP</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>TRUE</td>
<td>TYPE</td>
<td>UNLIKE</td>
<td>UNTIL</td>
</tr>
<tr>
<td>UPDATE</td>
<td>WAIT</td>
<td>WHERE</td>
<td>WHILE</td>
<td>WINFORM</td>
</tr>
<tr>
<td>XOR</td>
<td>YES</td>
<td>YRT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**What Can You Include in a Procedure?**

You can include the following items in a Maintain procedure:

- **Maintain language commands**, which are described in *Command Reference* on page 55.
All Maintain commands must be located within a Maintain function, except for the MAINTAIN, MODULE, DESCRIBE, CASE, and END commands, as well as global DECLARE commands, all of which must be located outside of a function. In the Procedure Editor, commands are displayed in blue by default.

- **Comments**, which are described in *Adding Comments* on page 24. In the Procedure Editor, comments are displayed in green by default.

- **Blank lines**, which you may wish to add to separate functions and other logic so that the procedure is easier for you to read.

If a Maintain procedure is a starting procedure (sometimes known as a root procedure), and it is not called by any other Maintain procedures, it can also contain Dialogue Manager commands preceding the MAINTAIN command. Dialogue Manager commands are described in the *Developing Reporting Applications* manual.

### Multi-Line Commands

You can continue almost all Maintain commands onto additional lines. The continued command can begin in any column, and can be continued for any number of lines.

The only exceptions are the TYPE command, which uses a special convention for continuing, and the beginning of the REPEAT command, which cannot be continued.

In the following example, all continued lines are indented for clarity:

```maintain
MAINTAIN FILES VideoTrk
    AND Movies

. .
. .
IF CustInfo.FocIndex GT 1
    THEN COMPUTE CustInfo.FocIndex = CustInfo.FocIndex - 1;
    ELSE COMPUTE CustInfo.FocIndex = CustInfo.FocCount;
```
Terminating Command Syntax

When you code a Maintain command, you terminate its syntax using one of the following:

- **A semicolon (;).** For most commands that can be terminated with a semicolon, the semicolon is optional. Even when it is optional, supplying it is recommended.

  **Coding suggestion:** Supplying optional semicolons is preferable, because if you omit them, when you invoke functions in that procedure you must do so using the COMPUTE or PERFORM commands. By supplying optional semicolons in a procedure, you can invoke functions more directly, by simply specifying their names. Supplying optional semicolons is also preferable because if you supply them in a procedure, you can code assignment statements more succinctly by omitting the COMPUTE keyword.

  For example, the following NEXT command, assignment statement, and invocation of the DisplayEditForm function are all terminated with semicolons:

  ```
  FOR ALL NEXT CustID INTO CustOrderStack;
  EditFlag = CustOrderStack().Status;
  DisplayEditForm();
  ```

- **An end keyword.** Some commands, such as BEGIN, CASE, and REPEAT, bracket a block of code. You indicate the end of the block by supplying the appropriate END keyword (for example, ENDBEGIN, ENDCASE, or ENDREPEAT).

  In the following example, the CASE command is terminated with an ENDCASE keyword:

  ```
  CASE UpdateAcct
  UPDATE SavingsAcct FROM TransactionStack;
  IF FocError NE 0 THEN TransErrorLog();
  ENDCASE
  ```

  Most commands use one of these methods (a semicolon or an end keyword) exclusively, as described for each command in *Command Reference* on page 55.

Adding Comments

By adding comments to a procedure you can document its logic, making it easier to maintain. You can place a comment virtually anywhere in a Maintain procedure. You can place a comment on its own line, at the end of a command, or even in the middle of a command. You can also place a comment in the middle of the procedure, at the very beginning of the procedure before the MAINTAIN command, or at the very end of the procedure following the END command. You can place any text within a comment.

There are two types of comments:
Stream comments, which begin with $* and end with *$. Maintain interprets everything between these two delimiters as part of the comment. A comment can begin on one line and end on another line, and can include up to 51 lines.

For example:

```
MAINTAIN
  $* This is a stream comment *$
  TYPE "Hello world"
  $* This is a second stream comment.
This is still inside the second comment!
  $* This is the end of the second comment *$
$* Document the TYPE statement--> *$ TYPE "Hello again!"; $* Goodbye
*$
END
```

Line comments, which begin with $$ or -* and continue to the end of the line. For example:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay;
  -* This entire line is a comment.
COMPUTE Pay.NewSal/D12.2;
...
END
```

You can also place a comment at the end of a line of code:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay; $$ Put root seg into a stack
COMPUTE Pay.NewSal/D12.2;
...
END
```

You can even place a comment at the end of a line containing a command that continues onto the next line:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay  -* Put root seg into a stack
WHERE Department IS 'MIS';
COMPUTE Pay.NewSal/D12.2;
...
END
```
You can include all types of comments in the same procedure:

```
MAINTAIN
    TYPE "Hello world"; -* This is a TYPE command
    $* This is a stream comment
        that runs onto a second line *
    $* Document the TYPE statement--> *$ TYPE "Hello again!"; $$ Goodbye
    ...
END
```

While Maintain uses the same comment characters (-*) as Dialogue Manager, it is only in a Maintain procedure that comments can be placed at the end of a line of code.
Expressions Reference

An expression enables you to combine variables, constants, operators, and functions in an operation that returns a single value. Expressions are used in a wide variety of Maintain commands. You can build increasingly complex expressions by combining simpler ones.

In this chapter:

- Types of Expressions You Can Write
- Writing Numeric Expressions
- Writing Date Expressions
- Writing Date-Time Expressions
- Writing Character Expressions
- Writing Logical Expressions
- Writing Conditional Expressions
- Handling Null Values in Expressions

Types of Expressions You Can Write

This section describes the types of expressions you can write in Maintain:

- **Numeric.** Use a numeric expression to perform a calculation on numeric constants or variables. For example, you can write an expression to compute the bonus for each employee by multiplying the current salary by the desired percentage as follows:

  ```plaintext
  COMPUTE Bonus = Curr_Sal * 0.05 ;
  ```

  A numeric expression returns a numeric value. For details, see *Writing Numeric Expressions* on page 29.

- **Date and time.** Use a date and time expression to perform a calculation that involves dates and/or times. For example, you can write an expression to determine when a customer can expect to receive an order by adding the number of days in transit to the date on which you shipped the order as follows:

  ```plaintext
  COMPUTE Delivery/MDY = ShipDate + 5 ;
  ```
There are two types of date and time expressions:

- **Date expressions**, which return a date or an integer that represents the number of days, months, quarters, or years between two dates. For details, see *Writing Date Expressions* on page 34.

- **Date-time expressions**. For details, see *Writing Date-Time Expressions* on page 40.

- **Character**. Use a character expression to manipulate alphanumeric or text constants or variables. For example, you can write an expression to extract the first initial from an alphanumeric field as follows:

  \[
  \text{COMPUTE First_Init/A1} = \text{MASK (First_Name, '9$$$$$$$$$')} ;
  \]

  A character expression returns a character value. For details, see *Writing Character Expressions* on page 45.

- **Logical**. Use a logical expression to determine whether a particular relationship between two values is true. A logical expression returns TRUE or FALSE. For details see *Writing Logical Expressions* on page 49.

- **Conditional**. Use a conditional expression to assign a value based on the result of a logical expression. A conditional expression returns a numeric or character value. For details see *Writing Conditional Expressions* on page 52.

**Reference: Usage Notes for Expressions**

- Expressions in Maintain cannot exceed 40 lines of text or use more than 16 IF statements.

- Expressions are self-terminating. You do not use a semicolon to indicate the end of an expression. Semicolons are used only to terminate commands.

**Expressions and Variable Formats**

When you use an expression to assign a value to a variable, make sure that you give the variable a format that is consistent with the value returned by the expression. For example, if you use a character expression to concatenate a first name and last name and assign it to the variable FullName, make sure you define the variable as character (that is, as alphanumeric or text).
Writing Numeric Expressions

A numeric expression performs a calculation that uses numeric constants, variables, operators, or functions to return a number. A numeric expression can consist of the following components, as highlighted below:

- A numeric constant. For example:
  ```
  COMPUTE COUNT/I2 = 1 ;
  ```

- A numeric variable. For example:
  ```
  COMPUTE RECOUNT/I2 = Count ;
  ```

- Two numeric constants or variables joined by a numeric operator. For example:
  ```
  COMPUTE BONUS/D12.2 = CURR_SAL * 0.05 ;
  ```

- A numeric function. For example:
  ```
  COMPUTE LONGEST_SIDE/D12.2 = MAX (WIDTH, HEIGHT) ;
  ```

- Two or more numeric expressions joined by a numeric operator. For example:
  ```
  COMPUTE PROFIT/D12.2 = (RETAIL_PRICE - UNIT_COST) * UNIT_SOLD ;
  ```

Reference: Numeric Operators

The following list shows the numeric operators you can use in an expression:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Integer division</td>
<td>DIV</td>
</tr>
<tr>
<td>Remainder division</td>
<td>MOD</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>**</td>
</tr>
</tbody>
</table>
Multiplication, DIV, MOD, and exponentiation are not supported for date expressions of any type. To isolate part of a date, use a simple assignment command.

**Syntax:** How to Use DIV: Integer Division

The DIV operator can be used in any valid expression to perform integer division. The result is an integer value and the remainder is truncated.

The syntax is:

```
expression DIV expression
```

**Example:** Using DIV to Perform Integer Division

In this example, the DIV operator is used to calculate the number of whole days that are equivalent to a number of hours:

```
COMPUTE Days/I4 = Hours DIV 24;
```

**Syntax:** How to Use MOD: Calculating the Remainder

The MOD operator can be used in any valid Maintain expression to calculate the remainder when division is performed.

The syntax is:

```
expression MOD divisor
```

The MOD operator always returns an integer value, and all decimal places are truncated.

**Example:** Using MOD to Calculate a Remainder

In the following example, the divisor is 10. The variables IntMod and DblMod contain the result.

```
MAINTAIN FILE Car
FOR 4 NEXT Country MPG INTO StkCar
REPEAT StkCar.FocCount Cnt/I4=1;
    COMPUTE IntMod/I4=StkCar(Cnt).MPG MOD 10;
    DblMod/D4.1=StkCar(Cnt).MPG MOD 10;
    TYPE "MPG=<<StkCar(Cnt).MPG" " IntMod=<<IntMod DblMod=<<DblMod"
ENDREPEAT Cnt=Cnt+1;
END
```
The decimal place in the variable DblMod is truncated, even though the format is D4.1.

\[
\begin{array}{ccc}
\text{MPG} & \text{INTMOD} & \text{DBLMOD} \\
16 & 6 & 6.0 \\
9 & 9 & 9.0 \\
11 & 1 & 1.0 \\
25 & 5 & 5.0 \\
\end{array}
\]

**Order of Evaluation**

Maintain performs numeric operations in the following order:

1. Exponentiation.
2. Division and multiplication.
3. Addition and subtraction.

When operators are at the same level, they are evaluated from left to right. Because expressions in parentheses are evaluated before any other expression, you can use parentheses to change this predefined order. For example, the following expressions yield different results because of parentheses:

\[
\begin{align*}
\text{COMPUTE PROFIT} &= \text{RETAIL_PRICE} - \text{UNIT_COST} \times \text{UNIT_SOLD} \\
\text{COMPUTE PROFIT} &= (\text{RETAIL_PRICE} - \text{UNIT_COST}) \times \text{UNIT_SOLD}
\end{align*}
\]

In the first expression, UNIT_SOLD is first multiplied by UNIT_COST, and the result is subtracted from RETAIL_PRICE. In the second expression, UNIT_COST is first subtracted from RETAIL_PRICE, and that result is multiplied by UNIT_SOLD.

**Evaluating Numeric Expressions**

Maintain follows a specific evaluation path for each numeric expression based on the format of the operands and the operators. If the operands all have the same format, most operations are carried out in that format. This is known as native-mode arithmetic. If the operands have different formats, Maintain converts the operands to a common format in a specific order of format precedence. Regardless of operand formats, some operators require conversion to specific formats so that all operands are in the appropriate format.

**Identical Operand Formats**

If all operands of a numeric operator are of the same format, you can use the following table to determine whether or not the operations are performed in that native format or if the operands are converted before and after executing the operation. In each case requiring conversion, operands are converted to the operational format and the intermediate result is returned in the operational format. If the format of the result differs from the format of the target variable, the result is converted to the format of the target variable.
### Writing Numeric Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operational Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+ Native</td>
</tr>
<tr>
<td>Subtraction</td>
<td>- Native</td>
</tr>
<tr>
<td>Multiplication</td>
<td>* Native</td>
</tr>
<tr>
<td>Full Division</td>
<td>/ Accepts single or double-precision floating point, converts all others to double-precision floating point.</td>
</tr>
<tr>
<td>Integer Division</td>
<td>DIV Native, except converts packed decimal to double-precision floating point.</td>
</tr>
<tr>
<td>Remainder Division</td>
<td>MOD Native, except converts packed decimal to double-precision floating point.</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>** Double-precision floating point.</td>
</tr>
</tbody>
</table>

#### Example: Identical Operand Formats

Because the following variables are defined as integers,

```plaintext
COMPUTE OperandOne/I4;
    OperandTwo/I4;
    Result/I4;
```

Maintain does the following multiplication in native-mode arithmetic (integer arithmetic):

```plaintext
COMPUTE Result = OperandOne * OperandTwo;
```

#### Different Operand Formats

If operands of a numeric operator have different formats, you can use the following table to determine what the common format is after Maintain converts them. Maintain converts the lower operand to the format of the higher operand before performing the operation.

<table>
<thead>
<tr>
<th>Order</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16-byte packed decimal</td>
</tr>
<tr>
<td>2</td>
<td>Double-precision floating point</td>
</tr>
<tr>
<td>3</td>
<td>8-byte packed decimal</td>
</tr>
<tr>
<td>Order</td>
<td>Format</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Single-precision floating point</td>
</tr>
<tr>
<td>5</td>
<td>Integer</td>
</tr>
<tr>
<td>6</td>
<td>Character (alphanumeric and text)</td>
</tr>
</tbody>
</table>

For example, if a 16-byte packed-decimal operand is used in an expression, all other operands are converted to 16-byte packed-decimal format for evaluation. On the other hand, if an expression includes only integer and alphanumeric operands, all alphanumeric operands are converted to integer format.

Maintain converts the alphanumeric to a numeric. If the alphanumeric is not a number, it is converted to 0 (zero), and 0 (zero) gets substituted into the equation.

If you assign a decimal value to an integer, Maintain truncates the fractional value.

**Continental Decimal Notation**

By default, you must use a decimal point to indicate a decimal position when writing a value in a Maintain procedure (for example, a COMPUTE statement), and a comma if you wish to demarcate thousands, regardless of the CDN setting.

To write the value in a procedure using the format matching the CDN setting for a value other than OFF (for example, ON, QUOTE, QUOTEP, SPACE), use MNTCON CDN_FEXINPUT ON in the EDASPROF file or user profile, and use quotation marks to delimit the value. You can use single or double quotes when CDN=ON or SPACE. However, you must use double quotes when CDN=QUOTE or QUOTEP. The MNTCON CDN_FEXINPUT command does not apply to values entered in Maintain forms at run time.

For more information on setting MNTCON CDN_FEXINPUT, see *MNTCON CDN_FEXINPUT* on page 129.

When entering values in forms at run time, observe the following rule:

- If CDN=OFF or QUOTEP, use a decimal point to enter decimal values.
- If CDN=ON, QUOTE or SPACE, use a comma to enter decimal values.

For more information on the SET CDN command, see the *Developing Reporting Applications* manual.
Writing Date Expressions

A date expression returns a date, a component of a date, or an integer that represents the number of days, months, quarters, or years between two dates.

A date expression can consist of the following components, highlighted below:

- A date constant. For example:

  ```
  COMPUTE StartDate/MDY= 'FEB 28 93';
  ```

  Note the use of single quotation marks around the date constant FEB 28 1993.

- A date variable. For example:

  ```
  COMPUTE NewDate = StartDate;
  ```

- An alphanumeric, integer, or packed variable with date edit options. For example, in the second COMPUTE command, OldDate is a date expression:

  ```
  COMPUTE OldDate/I6YMD = '980307';
  COMPUTE NewDate/YMD DFC 19 YRT 10 = OldDate;
  ```

- A calculation that uses addition, subtraction, or date functions to return a date. For example:

  ```
  COMPUTE Delivery/MDY = ShipDate + 5;
  ```

- A calculation that uses subtraction or date functions to return an integer (not a date) that represents the number of days, months, quarters, or years between two dates. For example:

  ```
  COMPUTE ResponseTime/I4 = ShipDate - OrderDate;
  ```

Formats for Date Values

Maintain enables you to work with dates in one of two ways:

- **In date format**, Maintain treats the value as a date. Date format interprets cross-century dates correctly, regardless of whether they are displayed with century digits. This is the preferred way of working with date values. The date is stored internally as an integer representing the number of days between the date and a standard base date. The base date is 12/31/1900 for all date variables declared in any operating environment using a 'D' for days, and also for all date variables declared in a Windows or UNIX environment using a 'Y' for years. The base date is 01/01/1901 for all date variables declared with a 'Y' in an S/390 environment.
In integer, packed, or alphanumeric format with date edit options, Maintain treats the value as an integer, a packed decimal, or an alphanumeric string. When displaying the value, Maintain formats it to resemble a date.

You can convert a date in one format to a date in another format simply by assigning one to the other. For example, the following assignment statements take a date stored as an alphanumeric variable formatted with date edit options and convert it to a date stored as a date variable:

```plaintext
COMPUTE AlphaDate/A6MDY = '120599';
RealDate/MDY = AlphaDate;
```

The following table illustrates how the format affects storage and display:

<table>
<thead>
<tr>
<th>Date Format</th>
<th>Integer, Packed, or Alphanumeric Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>For example: MDY</td>
<td>For example: A6MDY</td>
</tr>
<tr>
<td>Value</td>
<td>Stored</td>
</tr>
<tr>
<td>October 31, 1992</td>
<td>33542</td>
</tr>
<tr>
<td>November 01, 1992</td>
<td>33543</td>
</tr>
</tbody>
</table>

### Evaluating Date Expressions

The format of a variable determines how you can use it in a date expression. Calculations on dates in date format can incorporate numeric operators as well as numeric functions. If you need to perform calculations on dates in integer, packed, or alphanumeric format, we recommend that you first convert them to dates in date format, and then perform the calculations on the dates in date format.

Consider the following example, which calculates how many days it takes for your shipping department to fill an order by subtracting the date on which an item is ordered, OrderDate, from the date on which it is shipped, ShipDate:

```plaintext
COMPUTE TurnAround/I4 = ShipDate - OrderDate;
```

An item ordered on October 31, 1992 and shipped on November 1, 1992 should result in a difference of 1 day. The following table shows how the format affects the result:
If the date variables are in integer format, you can convert them to date format and then calculate TurnAround:

\[
\text{COMPUTE NewShipDate/MDY = ShipDate;} \\
\text{NewOrderDate/MDY = OrderDate;} \\
\text{TurnAround/I4 = NewShipDate - NewOrderDate;}
\]

**Selecting the Format of the Result Variable**

A date expression always returns a number. That number may represent a date or the number of days, months, quarters, or years between two dates. When you use a date expression to assign a value to a variable, the format you give to the variable determines how the result is displayed.

Consider the following commands. The first command calculates how many days it takes for your shipping department to fill an order by subtracting the date on which an item is ordered, ORDERDATE, from the date on which it is shipped, SHIPDATE. The second calculates a delivery date by adding 5 days to the date on which the order is shipped, SHIPDATE.

\[
\text{COMPUTE TURNAROUND/I4 = SHIPDATE - ORDERDATE ;} \\
\text{COMPUTE DELIVERY/MDY = SHIPDATE + 5 ;}
\]

In the first command, the date expression returns the number of days it takes to fill an order. Therefore, the associated variable, TURNAROUND, must have an integer format. In the second command, the date expression returns the date on which the item will be delivered. Therefore, the associated variable, DELIVERY, must have a date format.

**Manipulating Dates in Date Format**

This section provides additional information on how to write expressions using values represented in date format. It describes how to:

- Use a date constant in an expression.
- Extract a date component.
- Combine variables with different components in an expression.
Using a Date Constant in an Expression

When you use a date constant in a calculation with variables in date format, you must enclose it in single quotation marks, otherwise, Maintain interprets it as the number of days between the constant and the base date (December 31, 1900). The following example shows how to initialize STARTDATE with the date constant 02/28/93:

\[
\text{COMPUTE STARTDATE/MDY} = '022893';
\]

Extracting a Date Component

Date components include days, months, quarters, and years. You can write an expression that extracts a component from a variable in date format. The following example shows how you can extract a month from SHIPDATE, which has the format MDY:

\[
\text{COMPUTE SHIPMONTH/M = SHIPDATE} ;
\]

If SHIPDATE has the value November 23, 1992, the above expression returns the value 11 for SHIPMONTH. Note that calculations on date components automatically produce a valid value for the desired component. For example, if the current value of SHIPMONTH is 11, the following expression:

\[
\text{COMPUTE ADDTHREE/M = SHIPMONTH + 3} ;
\]

correctly returns the value 2, not 14.

You cannot write an expression that extracts days, months, or quarters from a date that did not have these components. For example, you cannot extract a month from a date in YY format, which represents only the number of years.

Combining Variables With Different Components in an Expression

When using variables in date format, you can combine variables with a different order of components within the same expression. For example, consider the following two variables, where DATE_PAID has the format YYMD and DUE_DATE has the format MDY. You can combine these two variables in an expression to calculate the number of days that a payment is late as follows:

\[
\text{COMPUTE DAYS_LATE/I4 = DATE_PAID - DUE_DATE} ;
\]
In addition, you can assign the result of a date expression to a variable with a different order of components from the variables in the expression. For example, consider the variable DATE_SOLD, which contains the date on which an item is sold, in YYMD format. You can write an expression that adds 7 days to DATE_SOLD to determine the last date on which the item can be returned, and then assign the result to a variable with DMY format, as in the following COMPUTE command:

```compute return_by/dmy = date_sold + 7 ;```

**Different Operand Date Formats**

In an expression in a procedure, all date formats are valid. If you have an expression that operates on date variables with different formats (for example, QY and MDY), Maintain converts one variable to the format of the other variable in order to perform the operation.

However, there are a few types of date variables that you cannot use in a mixed-format date expression. These variables, formatted as single components, such as a day of the week or year (formats D, W, Y, and YY), cannot be meaningfully converted to a more complete date (such as a year with a month). Of course, you can use these date variables in same-type date expressions.

If a date with format M is compared to a date with format Q (or vice versa), the operand on the right is converted to the format of the operand on the left, and then the comparison is performed.

For all other date-to-date comparisons, the date with the lesser format is promoted to the format of the higher date, where possible. If conversion is not possible, an error is generated.

The following conversion hierarchy applies to date formats:

<table>
<thead>
<tr>
<th>Order</th>
<th>Date Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dates with three components (for example, MDY, YYMD, Julian dates).</td>
</tr>
<tr>
<td>2</td>
<td>Dates with two components, one of which is a month (for example, MYY or YM).</td>
</tr>
<tr>
<td>3</td>
<td>Dates with two components, one of which is a quarter (for example, YQ).</td>
</tr>
<tr>
<td>4</td>
<td>Single component M or Q.</td>
</tr>
<tr>
<td>5</td>
<td>All other formats.</td>
</tr>
</tbody>
</table>

Dates in the fifth category do not generally get promoted.
When you have dates of two different types, dates in the lower category are promoted to the higher type.

**Using Addition and Subtraction in a Date Expression**

When performing addition or subtraction in a date expression:

- Adding a number to a date yields a date. It is up to you to make sure that the expression resolves to a meaningful value.

- Subtracting one date from another yields an integer that represents the difference between the two dates.

- When a date with format M or Q is subtracted from a higher type of date, the operand on the right is converted to the format of the operand on the left.

- When a two-component date is subtracted from a three-component date, or vice versa, the variable with the lesser format is promoted to the type of the variable with the higher format.

- When subtracting a Q format date from an M format date, or vice versa, the operand on the right is converted to the same format as the operand on the left.

- Subtracting a number from a date yields a date with the same format as the original date.

- You cannot subtract a date from a number, and you cannot add a date to a date.

**Example: Using Addition and Subtraction in a Date Expression**

Given the following variable definitions

```plaintext
DECLARE Days/D = 23;
DECLARE OldYear/YY = 1960;
DECLARE NewYear/YY = 1994;
DECLARE YearsApart/YY;
DECLARE OldYearMonth/YM = 9012;
DECLARE NewYearMonth/YM;
DECLARE FullDate/YMD = 870615;
```

the following COMPUTE commands are valid:

```plaintext
COMPUTE
  YearsApart = NewYear - OldYear;
  NewYear = OldYear + 2;
  NewYearMonth = OldYearMonth - FullDate;
```
However, the next series of COMPUTE commands are invalid, because they include date variables formatted as just a day (Days) or just a year (OldYear) in a mixed-format date expression:

```plaintext
COMPUTE
NewYear = FullDate - OldYear;
FullDate = OldYearMonth + Days;
```

**Writing Date-Time Expressions**

Date-time values for Maintain may be supplied in one of the following ways:

- As a value in a computed expression, enclosed in double or single quotes.
- As a value extracted or computed by a date-time function.
- Using an application Winform.

WebFOCUS Maintain supports the date-time data type with the following restrictions:

- The default date-time format separators (/) must be used. Other separators are not supported.
- When you create a WHERE statement or an IF THEN ELSE clause, you must use a variable as the test value.
- The format SET DATEFORMAT, used to change the default input format, is not supported.
- The SET commands WEEKFIRST and DTSTRICT are not supported.
- Computing an expression to DT (value) is not supported.

**Syntax:** **How to Write Date-Time Expressions**

A date-time constant in a Maintain procedure, and in an IF expression in a report procedure, has one of the following formats (note that in an IF expression, if the value contains no blanks or special characters, the single quotation marks are not necessary)

```plaintext
'date_string
[time_string]' 'time_string
[date_string]'
```
where:

\textit{time\_string}

Cannot contain blanks. Time components are separated by colons and may be followed by AM, PM, am, or pm. For example:

\begin{verbatim}
14:30:20:99       (99 milliseconds)
14:30
14:30:20.99       (99/100 seconds)
14:30:20.999999   (999999 microseconds)
02:30:20:500pm
\end{verbatim}

Note that 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 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.

\textit{date\_string}

Can have one of the following three formats:

- **Numeric string format** is exactly four, six, or eight digits. Four-digit strings are considered to be a year (century must be specified). The month and day are set to January 1. Six-digit and eight-digit strings contain two or four digits for the year, followed by two for the month, and then two for the day.

If a numeric-string format longer than eight digits is encountered, it is treated as a combined date-time string in the H\textit{nn} format described in the \textit{Describing Data With WebFOCUS Language} manual. The following are examples of numeric string date constants:

\begin{verbatim}
99
1999
19990201
\end{verbatim}

- **Formatted-string format** contains a one-digit or two-digit day, a one-digit or two-digit month, and a two-digit or four-digit year separated by spaces, slashes, hyphens, or periods. 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 \texttt{DATEFORMAT} setting. The following are examples of formatted-string date constants:

\begin{verbatim}
1999/05/20
5 20 1999
99.05.20
1999–05–20
\end{verbatim}
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 the day is 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. For example:

Jan 6 2000

**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.
- In each date format, two-digit years are interpreted using the [F]DEFCENT and [F]YRTHRESH settings.

**Example:** Using a Date-Time Value in a COMPUTE Command

```sql
COMPUTE RAISETIME/HYYMDIA = '20000101 09:00AM';
```

**Manipulating Date-Time Values Directly**

The only direct operations that can be performed on date-time variables and constants are comparison using a logical expression and simple assignment of the form `A = B`. All other operations are accomplished through a set of date-time subroutines. For more information see *Writing Character Expressions* on page 45.

**Comparing and Assigning Date-Time Values**

Any two date-time values can be compared, even if their lengths do not match.

If a date-time field supports missing values, fields that contain the missing value have a greater value than any date-time field can have. Therefore, in order to exclude missing values from report output when using a GT or GE operator in a selection test, it is recommended that you add the additional constraint `field` NE MISSING to the selection test:

```
date_time_field {GT|GE} date_time_value AND date_time_field NE MISSING
```

Assignments are permitted between date-time formats of equal or different lengths. Assigning a 10-byte date-time value to an 8-byte date-time value truncates the microsecond portion (no rounding takes place). Assigning a short value to a long one sets the low-order three digits of the microseconds to zero.
Other operations, including arithmetic, concatenation, and the reporting operators EDIT and LIKE on date-time operands are not supported. Reporting prefix operators that work with alphanumeric fields are supported.

**Example: Testing for Missing Date-Time Values**

Consider the DATETIM2 Master File:

```plaintext
FILE=DATETIM2, SUFFIX=FOC,$
SEGNAME=DATETIME, SEGTYPE=S0,$
FIELD=ID, ID, USAGE = I2,$
FIELD=DT1, DT1, USAGE=HYYMDS, MISSING=ON,$
```

Field DT1 supports missing values. Consider the following request:

```plaintext
TABLE FILE DATETIM2
PRINT ID DT1
END
```

The resulting report output shows that in the instance with ID=3, the field DT1 has a missing value:

```
ID  DT1
--  ---
1  2000/01/01 02:57:25
2  1999/12/31 00:00:00
3  .
```

The following request selects values of DT1 that are greater than 2000/01/01 00:00:00 and are not missing:

```plaintext
TABLE FILE DATETIM2
PRINT ID DT1
  WHERE DT1 NE MISSING AND DT1 GT DT(2000/01/01 00:00:00);
END
```

The missing value is not included in the report output:

```
ID  DT1
--  ---
1  2000/01/01 02:57:25
```

**Date-Time Subroutines**

The following subroutines allow you to manipulate date-time values:
### Function Name | Description
---|---
HCNVRT | Converts date-time values to alphanumeric format for use with operators, such as EDIT, CONTAINS, and LIKE.
HINPUT | Converts an alphanumeric string to a date-time value.
HADD | Increments date-time values by a specified number of units.
HDIFF | Returns the number of units of a specific date-time component between two date-time values.
HNAME | Extracts specified components of a date-time value and converts them to alphanumeric format.
HPART | Extracts a component of a date-time value in numeric format.
HSEPTPT | Inserts the numeric value of a specified component in a date-time field.
HMIDNT | Changes the time portion of a date-time field to midnight.
HDDATE | Extracts the date components from a date-time field and converts them to a date field.
HDTTM | Converts a date field to a date-time field with the time set to midnight.
HTIME | Extracts all of the time components from a date-time field and converts them to a number of milliseconds or microseconds in numeric format.
HGETC | Returns the current date and time in date-time format.

For more information on these functions, see the *Using Functions* manual.

**Reference:** Notes Regarding ISO Standard Date-Time Representations

International Standard ISO 8601 describes the standards for numeric representations of date and time. Some of the relevant standards and notes about their implementation follow:

- The international standard date notation is YYYY-MM-DD.
- The international standard for the first day of a week is Monday. You can use the WEEKFIRST parameter with WebFOCUS procedures to control the day used as the first day of the week by the date-time functions. However, WebFOCUS Maintain does not support this setting.
The standard specifies that week 1 of a year is the first week of the year that has a Thursday. Combined with the standard of Monday as day 1, this rule ensures that week 1 has at least four of its days in the specified year.

The following rules represent an extension to the standard in this implementation:

- Whatever day you choose for your WEEKFIRST setting, the date-time functions define week 1 as the first week with at least four days in the specified year.
- With these rules, it is possible for the first few days of January to fall in the week prior to week 1. The international standard considers these dates to be in week 53 of the previous year. However, the date-time functions return zero for the week component when it falls in the week prior to week 1.
- The international standard notation for the time of day is hh:mm:ss using the 24-hour system. However, the date-time data type and date-time functions allow you to use the 12-hour system.

Writing Character Expressions

A character expression returns an alphanumeric or text value.

A character expression can consist of the following components, highlighted below:

- An alphanumeric or text constant (that is, a character string enclosed in single or double quotation marks). For example:
  
  \[\text{COMPUTE STATE} = 'NY';\]

- An alphanumeric or text variable. For example:
  
  \[\text{COMPUTE AddressPartTwo} = \text{STATE};\]

- A function returning an alphanumeric or text result. For example:
  
  \[\text{COMPUTE INITIAL/A1} = \text{MASK(FIRSTNAME, '9'$$$$$$$$$$')}\]

- Two or more alphanumeric and/or text expressions combined into a single expression using the concatenation operator. For example:
  
  \[\text{COMPUTE TITLE/A19} = 'DR. ' || \text{LAST_NAME};\]
Concatenating Character Strings

You can write an expression to concatenate several alphanumeric and/or text values into a single character string. The concatenation operator takes one of two forms, as shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Represents</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Weak concatenation.</td>
<td>Preserves trailing spaces.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strong concatenation.</td>
</tr>
</tbody>
</table>

Evaluating Character Expressions

Any non-character expression that is embedded in a character expression is automatically converted to a character string.

A constant must be enclosed in single or double quotation marks. Whichever delimiter you choose, you must use the same one to begin and end the string. The ability to use either single or double quotation marks provides the added flexibility of being able to use one kind of quotation mark to enclose the string, and the other kind as data within the string itself.

The backslash (\) is the escape character. You can use it to:

- Include a delimiter of a string (for example, a single quote) within the string itself, as part of the value. Simply precede the character with a backslash (\'), and Maintain will interpret the character as data, not as the end-of-string delimiter.
- Include a backslash within the string itself, as part of the value. Simply precede the backslash with a second backslash (\\).
- Generate a line feed (for example, when writing a message to a file or device using the SAY command). Simply follow the backslash with the letter n (\n).

When the backslash is used as an escape character, it is not included in the length of the string.
Using Single and Double Quotation Marks in a Character Expression

Because you can define a character string using single or double quotation marks, you can use one kind of quotation mark to define the string and the other kind within the string, as in the following expressions:

COMPUTE LastName = "O'HARA";
COMPUTE Msg/A40 = 'This is a "Message"';

Using a Backslash Character (\) in a Character Expression

You can include a backslash (the escape character) within a string as part of the value by preceding it with a second backslash. For example, the following source code

COMPUTE Line/A40 = 'The characters \\
\' are interpreted as ""';
.
.
TYPE "Escape info:  <Line"
displays:

Escape info:  The characters \' are interpreted as '

When the backslash is used as an escape character, it is not included in the length of the string. For example, a string of five characters and one escape character fits into a five-character variable:

COMPUTE Word/A5 = 'Can\\t'

Specifying a Path in a Character Expression

A path may, depending on the operating system, contain backslashes (\). Because the backslash is the escape character for character expressions, if you specify a path that contains backslashes in an expression, you must precede each of the backslashes with a second backslash. For example:

MyLogo/A50 = "C:\\ibi_img\\AcmeLogo.gif";

Extracting Substrings and Using Strong and Weak Concatenation

The following example shows how to use the SUBSTR function to extract the first initial from a first name, and then use both strong and weak concatenation to produce the last name, followed by a comma, followed by the first initial, followed by a period:

First_Init/A1 = SUBSTR (First_Name, 1, 1);
Name/A19 = Last_Name || (', ' | First_Init | '.');
Suppose that First_Name has the value Chris and Last_Name has the value Edwards. The above request evaluates the expressions as follows:

1. The SUBSTR function extracts the initial C from First_Name.
2. The expression in parentheses is evaluated. It returns the value C.
3. Last_Name is concatenated to the string derived in step 2 to produce the following:

   Edwards, C.

   Note that while Last_Name has the format A15, strong concatenation suppresses the trailing spaces.

Variable-Length Character Variables

You can enable a character variable to have a varying length by declaring it either as text (TX) or as alphanumeric with a length of zero (A0). TX and A0 are equivalent.

Specifying a varying length provides several advantages:

- **Increased length.** The variable can be as long as 32,766 characters. A fixed-length character variable, by contrast, has a maximum of 256 characters.

- **Flexible logic.** Variable length enables you to declare one variable that can accept values of many different lengths (ranging from zero to 32,766 characters). Other alphanumeric variables, by contrast, are always of fixed length.

  The default value of a variable-length character variable is a string of length zero.

- **No padding.** If you assign a character string to a longer fixed-length alphanumeric variable, the variable pads the value of the string with spaces to make up the difference. If you assign the same string to a variable-length variable, it stores the original value without padding it with spaces.

  Of course, if you assign a string with trailing spaces to either a fixed-length or variable-length character variable, the variable preserves those trailing spaces.

- **Optimized memory usage.** The memory used by a variable-length character variable is proportional to its size. The shorter the value, the less memory is used.

Note that the characteristics of variable-length data source fields differ from those of temporary variables. When declaring a data source field, TX is supported for relational data sources, and has a maximum length of 4094 characters. A0 is not supported for data source fields. For information about data source text fields in WebFOCUS Maintain applications, see the *Describing Data With WebFOCUS Language* manual.
**Example:** Padding and Trailing Spaces in Character Variables

Variable-length character variables, unlike those of fixed length, never pad strings with spaces. For example, if you assign a string of 11 characters to a 15-character fixed-length alphanumeric variable, the variable pads the value with four spaces to make up the difference.

Illustrating this, the following source code

```
COMPUTE Name/A15 = 'Fred Harvey' ;
TYPE "<<Name End of string" ;
```

displays:

Fred Harvey End of string

If you assign the same string of 11 characters to a variable-length variable, the variable stores the original value without padding it. Illustrating this, the following source code, in which Name is changed to be of variable length (specified by A0)

```
COMPUTE Name/A0 = 'Fred Harvey' ;
TYPE "<<Name End of string" ;
```

displays:

Fred HarveyEnd of string

If you assign a string with trailing spaces to a variable (of either fixed or varying length), the variable preserves those spaces. Illustrating this, the following source code

```
COMPUTE Name/A0 = 'Fred Harvey    ' ;
TYPE "<<Name End of string" ;
```

displays:

Fred Harvey End of string

**Writing Logical Expressions**

A logical expression determines whether a particular condition is true. There are two kinds of logical expressions, relational and Boolean. The entities you wish to compare determine the kind of expression.

A relational expression returns TRUE or FALSE based on comparison of two individual values (either variables or constants). A Boolean expression returns TRUE or FALSE based on the outcome of two or more relational expressions.

You can use a logical expression to assign a value to a numeric variable. If the expression is true, the variable receives the value 1. If the expression is false, the variable receives the value 0.
Relational Expressions

A relational expression returns TRUE or FALSE based on the comparison of two individual values (either variables or constants). The following syntax lists the operators you can use in a relational expression:

character_expression char_operator character_constant
numeric_expression numeric_operator numeric_constant

where:

char_operator
Can be any of the following: EQ, NE, OMITS, CONTAINS.

numeric_operator
Can be any of the following: EQ, NE, LE, LT, GE, GT.

Boolean Expressions

Boolean expressions return a value of true (1) or false (0) based on the outcome of two or more relational expressions. Boolean expressions are often used in conditional expressions, which are described in Writing Conditional Expressions on page 52. You can also assign the result of a Boolean expression to a numeric or character variable, which will be set to 1 (if the expression is true) or 0 (if it is false). They are constructed using variables and constants connected by operators.

Syntax: How to Use Boolean Expressions

The syntax of a Boolean expression is:

(relational_expression) {AND|OR} (relational_expression)
NOT (logical_expression)

Boolean expressions can themselves be used as building blocks for more complex expressions. Use AND or OR to connect the expressions and enclose each expression in parentheses.

Evaluating Logical Expressions

If you assign a Boolean expression to a character variable, it may have the values TRUE, FALSE, 1, or 0. TRUE and 1 are equivalent, as are FALSE and 0. A numeric variable may have the values 1 or 0.

Alphanumeric constants with embedded blanks used in the expression must be enclosed in single quotation marks. An example is:

IF NAME EQ 'JOHN DOE'
OR cannot be used between constants in a relational expression. For example, the following expression is not valid

IF COUNTRY EQ 'US' OR 'BRAZIL' OR 'GERMANY'

Instead, it should be coded as a sequence of relational expressions:

IF (COUNTRY EQ 'US') OR (COUNTRY EQ 'BRAZIL') OR (COUNTRY EQ 'GERMANY')

Reference: Logical Operators

The following list shows the logical operators you can use in an expression:

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>EQ</td>
</tr>
<tr>
<td>Inequality</td>
<td>NE</td>
</tr>
<tr>
<td>Less than</td>
<td>LT</td>
</tr>
<tr>
<td>Greater than</td>
<td>GT</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>LE</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>GE</td>
</tr>
<tr>
<td>Contains the specified character string</td>
<td>CONTAINS</td>
</tr>
<tr>
<td>Omits the specified character string</td>
<td>OMITS</td>
</tr>
<tr>
<td>Negation</td>
<td>NOT</td>
</tr>
<tr>
<td>Conjunction</td>
<td>AND</td>
</tr>
<tr>
<td>Disjunction</td>
<td>OR</td>
</tr>
</tbody>
</table>

Boolean operators are evaluated after numeric operators from left to right in the following order of priority:

<table>
<thead>
<tr>
<th>Order</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EQ NE LE LT GE GT NOT CONTAINS OMITS</td>
</tr>
</tbody>
</table>
Writing Conditional Expressions

A conditional expression assigns a value based on the result of a logical expression. The assigned value can be numeric or character.

Syntax: How to Use Conditional Expressions

The syntax of a conditional expression is

\[
\text{IF boolean THEN \{expression1\} [ELSE \{expression2\} ]}
\]

where:

- \text{boolean} is a Boolean expression. Boolean expressions are described in \textit{Boolean Expressions} on page 50.
- \text{expression} is a numeric, character, date, or conditional expression.

When the Boolean expression is true, the conditional expression returns the THEN expression. Otherwise, it returns the ELSE expression if one is provided.

The THEN and ELSE expressions can themselves be conditional expressions. If the expression following THEN is conditional, it must be enclosed in parentheses. A conditional expression can have up to 16 IF statements.

The variable to which you assign the conditional expression must have a format compatible with the formats of the THEN and ELSE expressions.
Handling Null Values in Expressions

When data does not exist for a variable, Maintain assigns the following default value, depending on how the format of the variable has been defined:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default value without the MISSING attribute</th>
<th>Default value with the MISSING attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>zero</td>
<td>null</td>
</tr>
<tr>
<td>Date and time</td>
<td>space</td>
<td>null</td>
</tr>
<tr>
<td>Character</td>
<td>space</td>
<td>null</td>
</tr>
</tbody>
</table>

A null value (sometimes known as missing data) appears as a period (.) by default. You can change the character representation of the null value by issuing the SET NODATA command. For details, see the Developing Reporting Applications manual.

Null values affect the results of expressions that perform aggregating calculations such as averaging and summing. See the topics about null data and missing data in Assigning Null Values: The MISSING Constant on page 53 for information about the MISSING attribute in Master Files and the effect of null values in calculations.

Assigning Null Values: The MISSING Constant

You can assign the MISSING constant (that is, the null value) to variables (data source fields and temporary variables) that were defined with the MISSING attribute.

When you create a user-defined variable with the MISSING attribute and do not explicitly assign a value, it is created with the null value. For example, in the following command, Name is created with a null value:

```
COMPUTE Name/A15 MISSING ON = ;
```

Syntax: How to Assign Null Values: The MISSING Constant

The syntax for assigning a null value to an existing variable is:

```plaintext
COMPUTE target_variable = MISSING;
```
Assigning Null Values

Example: Suppose that the variable AcctBalance had been defined with the MISSING attribute. The command below assigns the null value to AcctBalance:

```
COMPUTE AcctBalance = MISSING;
```

Conversion in Mixed-Format Null Expressions

When a variable with a null value is assigned to a variable that is not defined with the MISSING attribute, the null value is converted to a zero or a space. For example, when the variable Q is assigned to R, the null value from Q is converted to a zero, because zero is the default value for numeric variables without the MISSING attribute.

```
Q/I4 MISSING ON = MISSING;
R/I4 = Q;
```

The same conversion occurs before any mathematical operations are applied if the variables are used as operands in arithmetic expressions.

Testing Null Values

You may test for the null value using comparison operators EQ or NE in an expression. You can test any variable that has been declared with the MISSING attribute. The null value is represented by the MISSING constant.

Syntax: How to Test Null Values

The syntax for testing whether a value is null is:

```
target_variable {EQ|NE} MISSING
```

Example: Testing Null Values

In this example, an IF command executes a BEGIN block if the variable Returns is null:

```
IF Returns EQ MISSING THEN BEGIN
  .
  .
  .
  ENDBEGIN
```
This reference provides a summary of the Maintain language commands and system variables, grouped by primary use. It also describes some commands that are outside the language but can be used to manage Maintain procedures. It then describes each command and system variable in detail.

When you develop a project, you can generate Maintain commands by:

- Using the Language Wizard in the Procedure Editor. The Wizard asks you questions about the logic you wish to create, and automatically generates the required commands.
- Coding the commands yourself in the Procedure Editor.

Language Summary

This topic summarizes all Maintain language commands, grouping them by their primary use (such as transferring control or selecting records). Each command and system variable is described in detail later in this chapter.

Defining a Procedure

The basic syntax consists of the commands that start and terminate a Maintain procedure. The commands are:

```
MAINTAIN

    Initiates the parsing and execution of a Maintain procedure. It is always the first line of the procedure.

END

    Terminates the execution of a Maintain procedure.
```

Defining a Maintain Function (a Case)

The following command defines Maintain functions:

```
CASE

    Defines a Maintain function. Maintain functions are also known as cases.
```
Defining Blocks of Code

The following command defines a block a code:

BEGIN

Defines a group of commands as a single block and enables you to issue them as a group. You can place a BEGIN block anywhere individual commands can appear.

Transferring Control

You can transfer control to another function within the current procedure, as well as to another procedure.

The commands that transfer control are:

PERFORM

Transfers control to another function. When the function finishes, control is returned to the command following PERFORM. You can also call a function directly, without PERFORM.

GOTO

Transfers control to another function or to a special label within the current function. When the function finishes, control does not return. You can also call a function directly, without GOTO.

CALL

Executes another Maintain procedure.

EXEC

Executes an external (non-Maintain) procedure.

Executing Procedures

The following commands run procedures, or prepare them for execution:

CALL

Executes a Maintain procedure, and enables you to pass data from the calling procedure.

EXEC

Executes a WebFOCUS procedure.

MNTCON COMPIL

Compiles a Maintain procedure to increase its execution speed. This command is outside the Maintain language, but is described in this chapter for your convenience.
MNTCON EX

Executes an uncompiled Maintain procedure. This command is outside the Maintain language, but is described in this chapter for your convenience.

MNTCON RUN

Executes a compiled Maintain procedure. This command is outside the Maintain language, but is described in this chapter for your convenience.

Encrypting Files

You can use the following commands to prevent unauthorized users from viewing the content of procedure files and Master Files.

ENCRYPT

Encodes procedure files and Master Files to prevent unauthorized users from viewing their content. This command is outside the Maintain language, but is described in this chapter for your convenience.

DECRYPT

Decodes files that have been encoded using the ENCRYPT command. This command is outside the Maintain language, but is described in this chapter for your convenience.

Using Loops

The following command supports looping:

REPEAT

Enables a circular flow of control.

Using Forms

The following command is responsible for presentation logic:

Winform

Displays a form by which end users can read, enter, and edit data, and manipulate control properties.

Defining Classes

The following command enables you to define classes:

DESCRIBE

Defines classes and data type synonyms.
Creating Variables

The following commands enable you to create variables:

**DECLARE**

Creates local and global variables, including objects.

**COMPUTE**

Creates global variables, including global objects. It can also assign values to existing variables.

Assigning Values

Maintain enables you to assign values to existing variables using the following command:

**COMPUTE**

Assigns values to existing variables.

Manipulating Stacks

Maintain provides several stack commands to manage the contents of stacks. Unless otherwise specified, each command operates on all rows in the stack. The following example copies the contents of the Indata stack to the Outdata stack:

**FOR ALL COPY FROM Indata INTO Outdata;**

One row or a range of rows may be specified to limit which rows are affected. As an example

**FOR 100 COPY FROM Indata(4) INTO Outdata;**

copies 100 records of the Indata stack, starting from the fourth record, and places them into the stack Outdata.

The stack commands are:

**COPY**

Copies data from one stack to another.

**STACK SORT**

Sorts data in a stack.

**STACK CLEAR**

Initializes a stack.

**INFER**

Defines the columns in a stack.
In addition, there are two variables associated with a stack which can be used to manipulate individual rows or groups of rows in the stack. The stack variables are:

FocCount
Is the number of rows in the stack.

FocIndex
Is a pointer to the current instance in the stack.

Selecting and Reading Records

The record selection commands retrieve data from the data source and change position in the data source.

The commands are:

NEXT
Starts at the current position and moves forward through the data source. NEXT can retrieve data from one or more rows.

MATCH
Searches the entire segment for a matching field value. It retrieves an exact match in the data source.

REPOSITION
Changes the data source position to be at the beginning of the chain.

In addition, there is a system variable that provides a return code for NEXT and MATCH:

FocFetch
Signals the success or failure of a NEXT or MATCH command.

You can use the following commands to directly interface with a DBMS:

SYS_MGR.PRE_MATCH
Turns off preliminary database operation checking before an update.

SYS_MGR.GET_PRE_MATCH
Determines whether prematch checking is on or off.

SYS_MGR.ENGINE
Passes SQL commands directly to a DBMS.
**SYS_MGR.DBMS_ERRORCODE**

Retrieves a DBMS return code after an operation.

**Conditional Actions**

The conditional commands are:

**IF**

Issues a command depending on how an expression is evaluated.

**ON MATCH**

Determines the action to take when the prior MATCH command succeeds.

**ON NOMATCH**

Defines the action to take if the prior MATCH fails.

**ON NEXT**

Defines the action to take if the prior NEXT command succeeds.

**ON NONEXT**

Defines the action to take if the prior NEXT command fails.

**Writing Transactions**

The commands that can be used to control transactions are:

**INCLUDE**

Adds one or more new data source records.

**UPDATE**

Updates the specified data source fields or columns. Can update one or more records at a time.

**REVISE**

Adds new records to the data source and updates existing records.

**DELETE**

Deletes one or more records from the data source.

**COMMIT**

Makes all data source changes since the last COMMIT permanent.
ROLLBACK

Cancels all data source changes made since the last COMMIT.

There are several system variables that you can use to determine the success or failure of a data source operation or an entire logical transaction:

**FocCurrent**

Signals the success or failure of a COMMIT or ROLLBACK command.

**FocError**

Signals the success or failure of an INCLUDE, UPDATE, REVISE, or DELETE command.

**FocErrorRow**

If an INCLUDE, UPDATE, REVISE, or DELETE command that writes from a stack fails, this returns the number of the row that caused the error.

You can use the following commands to directly interface with a DBMS:

**SYS_MGR.PRE_MATCH**

Turns off preliminary database operation checking before an update.

**SYS_MGR.GET_PRE_MATCH**

Determines whether prematch checking is on or off.

**SYS_MGR.ENGINE**

Passes SQL commands directly to a DBMS.

**SYS_MGR.DBMS_ERRORCODE**

Retrieves a DBMS return code after an operation.

**Setting WebFOCUS Server Parameters**

You can communicate with the WebFOCUS Server using the following commands:

**SET**

Sets WebFOCUS Server parameters. This command is outside the Maintain language, but is described in this chapter for your convenience.

**SYS_MGR.FOCSET**

Sets WebFOCUS Server parameters (without having to set them in EDASPROF).
Configuring a Shared Application Server

You can dramatically increase the speed of your applications if you set them up on a Shared Application Server. For more information on this technique, see the Developing WebFOCUS Maintain Applications manual.

The following commands set up your application to run under a Shared Application Server:

**MNTCON COMPILE**

Compiles a procedure to increase its execution speed. This command is outside the Maintain language, but is described in this chapter for your convenience.

**MNTCON LOADIMAGE**

Loads all of the Maintain procedures in a WebFOCUS Maintain application prior to running the application on a Shared Application Server. This command is outside the Maintain language, but is described in this chapter for your convenience.

**MNTCON RUNIMAGE**

Runs a Maintain procedure on a Shared Application Server. This command is outside the Maintain language, but is described in this chapter for your convenience.

Using Libraries of Classes and Functions

You can import libraries using the following command:

**MODULE**

Imports a library of shared class definitions or functions into a Maintain procedure.

Messages and Logs

You can write messages to files, consoles, and forms using the following commands:

**SAY**

Writes messages to a file or to the default output device.

**TYPE**

Writes messages to a file or a form.

In addition, there is a system stack that is automatically populated with messages posted to the default output device by Maintain procedures (except for the starting procedure) and external procedures:

**FocMsg**

Contains messages posted by Maintain and WebFOCUS procedures.
BEGIN

The BEGIN/ENDBEGIN construction enables you to issue a set of commands. Because you can use this construction anywhere an individual Maintain command can be used, you can use a set of commands where before you could issue only one command. For example, it can follow ON MATCH, ON NOMATCH, ON NEXT, ON NONEXT, or IF.

Syntax: How to Use the BEGIN Command

The syntax for the BEGIN command is

```
BEGIN
  command
  ...
  ...
ENDBEGIN
```

where:

- **BEGIN**
  Specifies the start of a BEGIN/ENDBEGIN block.

- **command**
  Is one or more Maintain commands, except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE. BEGIN blocks can be nested, allowing you to place BEGIN and ENDBEGIN commands between BEGIN and ENDBEGIN commands.

- **ENDBEGIN**
  Specifies the end of a BEGIN block.

Note: You cannot assign a label to a BEGIN/ENDBEGIN block of code or execute it outside the bounds of the BEGIN/ENDBEGIN construction in a procedure.

Example: BEGIN With ON MATCH

The following example illustrates a block of code that executes when MATCH is successful:

```
MATCH Emp_ID
  ON MATCH BEGIN
    COMPUTE Curr_Sal = Curr_Sal * 1.05;
    UPDATE Curr_Sal;
    COMMIT;
ENDBEGIN
```
**Example:** **BEGIN With ON NEXT**

This example shows BEGIN and ENDBEGIN with ON NEXT:

```plaintext
ON NEXT BEGIN
    TYPE "Next successful."
    COMPUTE New_Sal = Curr_Sal * 1.05;
    PERFORM Cleanup;
ENDBEGIN
```

**Example:** **BEGIN With IF**

You can also use BEGIN and ENDBEGIN with IF to run a set of commands depending on how an expression is evaluated. In the following example, BEGIN and ENDBEGIN are used with IF and FocError to run a series of commands when the prior command fails:

```plaintext
IF FocError NE 0 THEN BEGIN
    TYPE "There was a problem."
    .
    .
ENDBEGIN
```

**Example:** **Nested BEGIN Blocks**

The following example nests two BEGIN blocks. The first one starts if there is a MATCH on Emp_ID and the second starts if UPDATE fails:

```plaintext
MATCH Emp_ID FROM Emps(Cnt);
ON MATCH BEGIN
    TYPE "Found employee ID <Emps(Cnt).Emp_ID";
    UPDATE Department Curr_Sal Curr_JobCode Ed_Hrs
    FROM Emps(Cnt);
    IF FocError GT 0 THEN BEGIN
        TYPE "Was not able to update the data source."
        PERFORM Errorhnd;
        ENDBEGIN
    ENDBEGIN
ENDBEGIN
```

**CALL**

Use the CALL command when you need one procedure to call another. When you use CALL, both the calling and called procedures communicate using variables: local variables that you pass between them and the global transaction variables FocError, FocErrorRow, and FocCurrent. CALL allows you to link modular procedures, so each procedure can perform its own set of discrete operations within the context of your application. Since called procedures can reside on different servers, you can physically partition applications across different platforms.
For additional information about requirements for passing variables, see the *Developing WebFOCUS Maintain Applications* manual.

**Syntax:** How to Use the CALL Command

The syntax of the CALL command is:

```
CALL
  procedure [AT server ] [KEEP|DROP] [PATH {VAR|LIST}] [FROM var_list]
  [INTO var_list]

[;]
  var_list: {variable} [{variable} ...]
```

where:

- **procedure**
  
  Is the name of the Maintain procedure to run.

- **AT server**
  
  Identifies the server of the called procedure if the called procedure and calling procedure are on different servers.

  In most situations the Maintain Development Environment supplies this phrase during the deployment phase and you should refrain from coding it yourself. If you do code the AT server clause in your procedure, it will not change during deployment. You can use a variable to supply the server name.

- **KEEP|DROP**
  
  The DROP parameter terminates the server session that the AT server phrase creates. The KEEP parameter leaves the server session active for reuse by subsequent calls. KEEP is the default value.

- **PATH**
  
  Is used to specify additional locations (search paths) the system should use when searching for dependent resources (Master Files, imported modules, and others). The path location names are application names existing within the APPROOT directory structure or application names that have been introduced with the APP MAP command. The search path value can be in the form of a Maintain variable or a list of literal values enclosed in quotes, as follows:

```
CALL Procedure AT server PATH "AppDir1 AppDir2 AppDir3" ;
CALL Procedure AT server PATH MyVariable ;
```
FROM

Is included if this Maintain procedure passes one or more variables to the called procedure.

INTO

Is included if the called Maintain procedure passes one or more variables back to this procedure.

var_list

Are the scalar variables and stacks that are passed to or from this procedure. Multiple variables are separated by blank spaces.

variable

Is the name of a scalar variable or stack. You can pass any variable except for those defined as variable-length character (that is, those defined as A0 or TX) and those defined using STACK OF.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.

Example: Calling Procedures to Validate Data

The following example shows three Maintain procedures. The first displays a form to collect employee IDs and salaries. It then calls Validate to make sure that the salaries are in a range. If they are all valid, it calls PutData and includes them in the data source. If not, it sets FocError to the invalid row and redispays the data.

MAINTAIN FILE EMPLOYEE
INFER EMP_ID CURR_SAL INTO EMPSTACK;
Winform Show EMPL;

CASE VALIDATE_DATA
CALL VALIDATE FROM EMPSTACK;
IF FOCERROR EQ 0 THEN BEGIN
   CALL PUTDATA FROM EMPSTACK;
   TYPE "DATA ACCEPTED";
ENDBEGIN

ELSE BEGIN
   TYPE "THERE WAS AN ERROR IN ROW <FOCERROR";
   TYPE "TRY AGAIN";
ENDBEGIN
ENDCASE
END
The Validate procedure contains:

```plaintext
MAINTAIN FILE EMPLOYEE FROM EMPSTACK
INFER EMP_ID INTO EMPSTACK;
COMPUTE CNT/I4=1;
REPEAT EMPSTACK.FOCCOUNT;
    IF EMPSTACK(CNT).CURR_SAL GT 100000 THEN BEGIN
        COMPUTE FOCERROR=CNT;
        GOTO EXITREPEAT;
    ENDBEGIN
    ELSE COMPUTE CNT=CNT+1;
ENDREPEAT
END
```

The PutData procedure, residing on a remote WebFOCUS Server, contains:

```plaintext
MAINTAIN FILE EMPLOYEE FROM EMPSTACK
INFER EMP_ID INTO EMPSTACK;
FOR ALL INCLUDE EMP_ID CURR_SAL FROM EMPSTACK;
END
```

**Example:** Using a Variable Server Name

The following syntax demonstrates the use of the variable MYSERVE passing the server name EDASERVE in the CALL (or EXEC) statement.

```plaintext
COMPUTE MYSERVE/A8='EDASERVE';
....
CALL CALLPROC AT MYSERVE (...path ....)
```

To disable this feature, see the server profile command `MNTCON REMOTESTYLE` on page 137.
**Example:**  **Calling Procedures to Populate Stacks**

The following example shows all of the models and body types for the displayed country and car. The first calls GETCARS to populate the stack containing Country and Car. Maintain then calls GETMODEL to populate the other stack with the proper information. Each time a new Country and Car combination is introduced, Maintain calls GETMODEL to repopulate the stack.

```
MAINTAIN FILE CAR
INFER COUNTRY CAR INTO CARSTK;
INFER COUNTRY CAR MODEL BODYTYPE INTO DETSTK;
CALL GETCARS INTO CARSTK;
PERFORM GET_DETAIL;
Wiform Show CARFORM;

CASE GET_DETAIL
CALL GETMODEL FROM CARSTK INTO DETSTK;
ENDCASE

CASE NEXTCAR
IF CARSTK.FOCINDEX LT CARSTK.FOCCOUNT
    THEN COMPUTE CARSTK.FOCINDEX= CARSTK.FOCINDEX +1;
    ELSE COMPUTE CARSTK.FOCINDEX = 1;
PERFORM GET_DETAIL;
ENDCASE

CASE PREVCAR
IF CARSTK.FOCINDEX GT 1
    THEN COMPUTE CARSTK.FOCINDEX= CARSTK.FOCINDEX -1;
    ELSE COMPUTE CARSTK.FOCINDEX = CARSTK.FOCCOUNT;
PERFORM GET_DETAIL;
ENDCASE
```

The procedure GETCARS loads all Country and Car combinations into CARSTK.

```
MAINTAIN FILE CAR INTO CARSTK
FOR ALL NEXT COUNTRY CAR INTO CARSTK;
END
```

The procedure GETMODEL loads all model and body type combinations into CARSTK for displayed Country and Car combinations.

```
MAINTAIN FILE CAR FROM CARSTK INTO DETSTK
INFER COUNTRY CAR INTO CARSTK;
STACK CLEAR DETSTK;
REPOSITION COUNTRY;
FOR ALL NEXT COUNTRY CAR MODEL BODYTYPE INTO DETSTK
    WHERE COUNTRY EQ CARSTK(CARSTK.FOCINDEX).COUNTRY
        AND CAR   EQ CARSTK(CARSTK.FOCINDEX).CAR;
END
```
CASE

The CASE command allows you to define a Maintain function. (Maintain functions are sometimes also called cases.) The CASE keyword defines the beginning of the function, and the ENDCASE keyword defines its end.

You can pass values to a Maintain function using its parameters, and you can pass values from a Maintain function using its parameters and its return value.

You can call a Maintain function in one of the following ways:

- Issuing a PERFORM or GOTO command.
- Calling the function directly.
- Calling the function as an event handler.
- Calling the function using the IWCTrigger JavaScript® or VBScript function.

Once control has branched to the function, it proceeds to execute the commands within it. If control reached the end of the function (that is, the ENDCASE command), it returns or exits depending on how the function was called:

- **Branch and return.** If the function was called by a branch-and-return command (that is, by a PERFORM command or an event handler), or called directly, control returns to the point immediately following the PERFORM, event handler, or function reference.

- **Branch.** If the function was called by a simple branch command (that is, by a GOTO command or by the IWCTrigger JavaScript or VBScript function), and control reaches the end of the function, it means that you have not provided any logic to direct control elsewhere and so it exits the procedure. If this is not the result you want, simply call the function using PERFORM instead of GOTO, or else issue a command before ENDCASE to transfer control elsewhere.

A CASE command that is encountered in the sequential flow of a procedure is not executed.

You assign a unique name to each function using the CASE command.
**Syntax:**

How to Use the CASE Command

The syntax for the CASE command is:

```sql
CASE functionname [TAKES p1/t1[,]...,[pn/tn]] [RETURNS result/t] [;]
   [declarations]
   commands
   .
   .
ENDCASE
```

where:

- **functionname**
  - Is the name you give to the function, and it can be up to 66 characters long. The name must begin with a letter, and can include any combination of letters, digits, and underscores (_).

- **TAKES p1/t1**
  - Specifies that the function takes parameters. `p1/t1...pn/tn` defines the parameters of the function (`p`) and the data type of each parameter (`t`). When you call the function, you pass it variables or constants to substitute for these parameters. Parameters must be scalar; they cannot be stacks.

  If the function is the Top function or is being used as an event handler, it cannot take parameters.

- **RETURNS result/t**
  - Specifies that the function returns a value. `result` is the name of the variable being returned, and `t` is the data type of the variable. The return value must be scalar. It cannot be a stack.

  If the function is the Top function or is being used as an event handler, it cannot return a value.

- **declarations**
  - Is an optional DECLARE command to declare any variables that are local to the function. These declarations must precede all other commands in the function.

- **commands**
  - Is one or more commands, except for CASE, DESCRIBE, END, MAINTAIN, and MODULE.
Terminates the parameter and return variable definitions of the CASE command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.

**Reference:** Usage Notes for CASE

- The first function in a procedure must be an explicit or implicit Top function.
- CASE commands cannot be nested.

**Reference:** Commands Related to CASE

- **PERFORM** transfers control to another function. When control reaches the end of the function, it returns to the command following PERFORM.

  - **GOTO** transfers control to another function or to the end of the current function. Unlike the PERFORM command, it does not return the control of the command that called the function.

**Calling a Function: Flow of Control**

When a function is called, and control in the function is complete, control returns to the next command after the call.

When the Increase function in the following example is complete, processing resumes with the line after the PERFORM command, the TYPE command:

```
PERFORM Increase;
TYPE "Returned from Increase";
.
.
.
CASE Increase
COMPUTE Salary = Salary * 1.05;
.
.
ENDCASE
```

**Passing Parameters to a Function**

In general, the parameters of a Maintain function are both input and output parameters:

- When one function calls another, the calling function passes the current values of the parameters.
When the called function terminates, it passes back the current values of the parameters. If the called function changes the values of any of its parameters, when it returns control to the calling function, the parameter variables in the calling function are set to those new values. The parameters are global to the calling and called functions.

This method of passing parameters is known as a call by reference, because the calling function passes a reference to the parameter variable (specifically, its address), not a copy of its value.

**Note:** There is one exception to this behavior. If you declare a function parameter (in the Function Editor or a CASE command) with one data type, but at run time you pass the function a value of a different data type, the value of the parameter is converted to the new data type. Data types, in this context, refer to basic data types such as fixed-length character (An where n is greater than zero), variable-length character (A0), text, date, date-time, integer, single-precision floating point, double-precision floating point, 8-byte packed decimal, and 16-byte packed decimal. Other data attributes, such as length, precision, MISSING, and display options, can differ without causing a conversion. Any changes that the called function makes to the value of the parameter will not get passed back to the calling function. The parameter is local to the called function.

This method of passing parameters is known as a call by value, because the calling function passes a copy of the value of the parameter variable, not a pointer to the actual parameter variable itself.

Note that you should not pass a constant as a function parameter if the function may change the value of that parameter.
Using the Return Value of a Function

If a function returns a value using the RETURNS phrase, you can call that function anywhere you can use an expression. For example:

```
MAINTAIN FILE HousePlan

COMPUTE ConferenceRoom/D6.2 = FindArea(CRlength,CRwidth);

CASE FindArea TAKES Length/D6.2, Width/D6.2 RETURNS Area/D6.2;
    Area = Length * Width;
ENDCASE

END
```

Using the Top Function

When you run a Maintain procedure, the procedure begins by executing its Top function. Every Maintain procedure has a Top function. Top does not take or return parameters. You can choose to define the Top function, beginning it with a CASE command and ending it with an ENDCASE command, as all other Maintain functions are defined. This is the recommended method for defining Top, and is how the Maintain Development Environment generates Top when creating a new procedure.

For example:

```
CASE Top
    .
    .
ENDCASE
```

COMMIT

The COMMIT command processes a logical transaction. A logical transaction is a group of data source operations in an application that are treated as one. The COMMIT operation signals a successful end of a transaction and writes the INCLUDE, UPDATE, and DELETE operations to the data source. The data source is (or should be) in a consistent state and all of the updates made by that transaction are now made permanent.
Syntax: How to Use the COMMIT Command

The syntax of the COMMIT command is

`COMMIT [;]`

where:

`;`

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Language Rules Reference on page 17.

Reference: Usage Notes for COMMIT

- When you issue a transaction that writes to multiple types of data sources, each DBMS evaluates its part of the transaction independently. When a COMMIT command ends the transaction, the success of the COMMIT against each data source type is independent of the success of the COMMIT against the other data source types.

  For example, if you run a procedure that accesses the FOCUS data sources Employee and JobFile and the SQL Server data source Salary, the success or failure of the COMMIT for Salary is independent of the success of the COMMIT for Employee and JobFile. This is known as a broadcast commit.

- COMMIT is automatically issued when a procedure does not contain any COMMIT commands, and the application is exited normally. This means an error did not cause program termination. If a procedure does not contain any COMMIT commands and it is terminated abnormally (for example if the system has run out of memory), a COMMIT is not issued. When a called procedure is exited, an automatic COMMIT is not issued. COMMIT is only issued when exiting the application.

- The variable FocCurrent is set after a COMMIT finishes. If the COMMIT is successful, FocCurrent is set to zero. If FocCurrent is not zero, the COMMIT failed, and all of the records in the logical unit of work will be rolled back because an internal ROLLBACK will be issued.

COMPUTE

The COMPUTE command enables you to:

- Create a global variable (including global objects), and optionally assign it an initial value. You can use the DECLARE command to create both local and global variables. See Local and Global Declarations on page 86 for more information about local and global variables.

- Assign a value to an existing variable.
Syntax: How to Use the COMPUTE Command

The syntax of the COMPUTE command is:

```plaintext
[COMPUTE]
target_variable[/datatype [DFC cc YRT yy] [missing]] [= expression];
.
.
missing: [MISSING {ON|OFF} [NEEDS] [SOME|ALL] [DATA]]
```

where:

**COMPUTE**

Is an optional keyword. It is required if the preceding command can take an optional semicolon terminator, but was coded without one. In all other situations it is unnecessary.

When the COMPUTE keyword is required, and there is a sequence of COMPUTE commands, the keyword needs to be specified only once for the sequence, for the first command in the sequence.

**target_variable**

Is the name of the variable which is being created and/or to which a value is being assigned. A variable name must start with a letter and can only contain letters, numbers and underscores (_).

**datatype**

Is included in order to create a new variable. If you are creating a simple variable, you can specify all built-in formats and edit options (except for TX) as described for the Master File FORMAT attribute in the Describing Data With WebFOCUS Language manual. If you are creating an object, you can specify a class. You must specify a data type when you create a new variable. You can only specify the data type of a variable once, and you cannot redefine the data type of an existing variable.

**DFC cc**

Specifies a default century that will be used to interpret any dates with unspecified centuries in expressions assigned to this variable. cc is a two-digit number indicating the century (for example, 19 would indicate the twentieth century). If this is not specified, it defaults to 19.

Specifying DFC cc is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

For information about working with cross-century dates, see Unique FOCUS Topics on the WebFOCUS Documentation CD.
YRT yy
Specifications a default threshold year for applying the default century identified in DFC cc. yy is a two-digit number indicating the year. If this is not specified, it defaults to 00.

When the year of the date being evaluated is less than the threshold year, the century of the date being evaluated defaults to the century defined in DFC cc plus one. When the year is equal to or greater than the threshold year, the century of the date being evaluated defaults to the century defined in DFC cc.

Specifying YRT yy is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

For information about working with cross-century dates, see Unique FOCUS Topics on the WebFOCUS Documentation CD.

missing
Is used to allow or disallow null values. This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

MISSING
If the MISSING syntax is omitted, the default value of the variable is zero for numeric variables and a space for character and date and time variables. If it is included, its default value is null.

ON
Sets the default value to null.

OFF
Sets the default value to zero or a space.

NEEDS
Is an optional keyword that clarifies the meaning of the command for a reader.

SOME
Indicates that for the target variable to have a value, some (at least one) of the variables in the expression must have a value. If all of the variables in the expression are null, the target variable will be null. This is the default.

ALL
Indicates that for the target variable to have a value, all the variables in the expression must have values. If any of the variables in the expression are null, the target variable will be null.
DATA
Is an optional keyword that clarifies the meaning of the command for a reader.

= Is optional when COMPUTE is used solely to establish format. The equal sign is required when expression is used.

expression
Is any standard Maintain expression, as defined in Expressions Reference on page 27. Each expression must be terminated with a semicolon (;). When creating a new variable using a class data type, you must omit expression.

Example: Moving the COMPUTE Keyword
You can place an expression on the same line as the COMPUTE keyword, or on a different line, so that

COMPUTE
TempEmp_ID/A9 = '000000000';

is the same as:

COMPUTE TempEmp_ID/A9 = '000000000';

Example: Multi-Statement COMPUTE Commands
You can type a COMPUTE command over as many lines as you need. You can also specify a series of assignments as long as each expression is ended with a semicolon. For example:

COMPUTE TempEmp_ID/A9 = '000000000';
    TempLast_Name/A15;
    TempFirst_Name/A10;

Example: Combining Several Statements Into One Line
Several expressions can be placed on one line as long as each expression ends with a semicolon. The following shows two COMPUTE expressions on one line and a third COMPUTE on the next line. The first computes a five percent raise and the second increases education hours by eight. The third concatenates two name fields into one field:

COMPUTE Raise/D12.2=Curr_Sal*1.05; Ed_Hrs=Ed_Hrs+8;
Name/A25 = First_Name || Last_Name;
Reference: Usage Notes for COMPUTE

- If the names of incoming data fields are not listed in the Master File, they must be defined before they can be used. Otherwise, rejected fields are unidentified and the procedure is terminated.

There are two different ways these fields can be defined. The first uses the notation:

```
COMPUTE target_variable/format =;
```

Because there is no expression after the equal sign (=), the field and its format is made known, but nothing else happens. If this style is used for a field in a form, the field appears on the form without a default value. Because COMPUTE is used solely to establish format, the equal sign is optional and the following syntax is also correct:

```
COMPUTE target_variable/format;
```

The second method of defining a user-defined field can be used when an initial value is desired. The syntax is:

```
COMPUTE target_variable/format = expression;
```

- Each field referred to or created in a Maintain procedure counts as one field toward the 3,072 field limit, regardless of how often its value is changed by COMPUTE commands. However, if a data source field is read by a Winform command and also has its value changed by a COMPUTE command, it counts as two fields.

Reference: Commands Related to COMPUTE

- **DEFINE** is a Master File attribute (not a command) that defines temporary fields and derives their values from other fields in the data source. This type of temporary field is called a virtual field. DEFINE automatically creates a corresponding virtual column in every stack that includes the segment of the field. For more information, see the *Describing Data With WebFOCUS Language* manual.

- **DECLARE** creates local and global variables.
**Using COMPUTE to Call Functions**

When you call a function as a separate statement (that is, outside of a larger expression), if the preceding command can take an optional semicolon terminator, but was coded without one, you must call the function in a COMPUTE or PERFORM command. You can use PERFORM for Maintain functions only, though not for Maintain functions that return a value. For example, in the following source code, the NEXT command is not terminated with a semicolon, so the function that follows it must be called in a COMPUTE command:

```plaintext
NEXT CustID INTO CustStack
COMPUTE VerifyCustID();
```

However, in all other situations, you can call functions directly, without a COMPUTE command. For example, in the following source code, the NEXT command is terminated with a semicolon, so the function that follows it can be called without a COMPUTE command:

```plaintext
NEXT CustID INTO CustStack;
VerifyCustID();
```

For more information about terminating commands with semicolons, see *Terminating Command Syntax* on page 24.

**COPY**

The COPY command copies some or all of the rows of one stack into another stack. You can use the COPY command to overwrite existing rows in the target stack, to add new rows, or to create the entire target stack.

You must define the contents of a stack before copying data into it. This can be accomplished by issuing a NEXT or an INFER command for data source fields, and COMPUTE for non-data source fields.

Source and target database stacks used in the Copy command must be derived from the same data source description. The COPY command checks that the data source and segment names are the same, and copies all columns in the source stack whose names and data types exactly match columns in the target stack. In this context, data type refers to the basic data type (such as integer) and all other data attributes including length, precision, null (MISSING), and display options such as zero suppression. Source and target columns do not need to be in the same sequence.

**Syntax:**  
How to Use the COPY Command

The syntax of the COPY command is

```plaintext
[FOR {int|ALL}|STACK] COPY FROM {stk[(row)]|CURRENT} INTO {stk[(row)]|CURRENT} [WHERE expression] [;]
```
where:

FOR

Is a prefix used with int or ALL to specify the number of rows to copy from the source (FROM) stack into the target (INTO) stack. If you omit both FOR and STACK, only the first row of the source stack is copied.

int

Is an integer expression that specifies how many source stack rows to copy into the target stack. If int exceeds the number of source stack rows between the starting row and the end of the stack, all of those rows are copied.

ALL

Indicates that all of the rows starting with either the first row or the subscripted row are copied from the source (FROM) stack into the target (INTO) stack.

STACK

Is a synonym for the prefix FOR ALL. If you omit both FOR and STACK, only the first row of the source stack is copied.

FROM

Is used with a stack name to specify which stack to copy the data from.

INTO

Is used with a stack name to specify the stack to be created or modified.

stk

Is the name of the source or target stack. You can specify the same stack as the source and target stacks.

row

Is a stack subscript that specifies a starting row number. It can be a constant, an integer variable or any Maintain expression that results in an integer value. If you omit row, it defaults to 1.

CURRENT

Specifies the Current Area. If you specify CURRENT for the source stack, all Current Area fields that also exist in the target stack are copied to the target stack. You cannot specify CURRENT if you specify FOR or STACK.

WHERE

Specifies selection criteria for copying stack rows. If you specify a WHERE phrase, you must also specify a FOR or STACK phrase.
expression

Is any Maintain expression that resolves to a Boolean expression. Unlike an expression in the WHERE phrase of the NEXT command, it does not need to refer to a data source field.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.

Example: Copying All Rows of a Stack
The following example copies the entire Emp stack into a new stack called Newemp:

FOR ALL COPY FROM Emp INTO Newemp;

Example: Copying a Specified Number of Stack Rows
The following example copies 100 rows from the Emp stack starting with row number 101. The rows are inserted beginning with row one of the stack Subemp:

FOR 100 COPY FROM Emp(101) INTO Subemp;

Example: Copying the First Row of a Stack
The following example copies the first row of the Emp stack into the first row in the Temp stack. Only the first row in the source (FROM) stack is copied because this is the default when a prefix is not specified for the COPY command. The data is copied into the first row of the Temp stack because the first row is the default when a row number is not supplied for the target (INTO) stack:

COPY FROM Emp INTO Temp;

Example: Copying a Row Into the Current Area
The following example copies the tenth row of the Emp stack into the Current Area. Only one row is copied from the Emp stack because the COPY command does not have a prefix. Every column in the stack is copied into the Current Area. If there is already a field in the Current Area with the same name as a column in the stack, the Current Area variable is replaced with data from the Emp stack:

COPY FROM Emp(10) INTO CURRENT;
Example: Copying Rows Based on Selection Criteria

You can also copy selected rows based on selection criteria. The following example copies every row in the World stack that has a Country equal to USA into a new stack called USA:

```
FOR ALL COPY FROM World INTO USA WHERE Country EQ 'USA';
```

The following takes data from one stack and places it into three different stacks: one to add data, one to change data, and one to update data.

```
FOR ALL COPY FROM Inputstk INTO Addstk WHERE Flag EQ 'A';
FOR ALL COPY FROM Inputstk INTO Delstk WHERE Flag EQ 'D';
FOR ALL COPY FROM Inputstk INTO Chngstk WHERE Flag EQ 'C';
FOR ALL INCLUDE Dbfield FROM Addstk;
FOR ALL DELETE Dbfield FROM Delstk;
FOR ALL UPDATE Dbfield1 Dbfield2 FROM Chngstk;
```

Example: Appending One Stack to Another

The following example takes an entire stack and adds it to the end of an existing stack. The subscript consists of an expression. Yeardata.FocCount is a stack variable where Yeardata is the name of the stack and FocCount contains the number of rows currently in the stack. By adding one to FocCount, the data is added after the last row:

```
FOR ALL COPY FROM Junedata INTO Yeardata(Yeardata.FocCount+1);
```

Reference: Usage Notes for COPY

- If the FOR int prefix specifies more rows than are in the source (FROM) stack, all of the rows are copied.
- Only the first row of the source (FROM) stack is copied if the COPY command does not include FOR.
- The entire stack is copied if the source (FROM) stack is not subscripted and FOR ALL is used.
- The row to start copying from defaults to the first row unless the source (FROM) stack is subscripted. If the source (FROM) stack is subscripted, the copy process starts with the row number and copies as many rows as specified in the FOR n prefix, or the remainder of the stack if FOR ALL is specified.
- No change is made to the source (FROM) stack unless it is also the target (INTO) stack.
- INTO CURRENT cannot be used with the FOR phrase and generates an error if specified.
- To copy an entire stack, specify FOR ALL without a subscripted source (FROM) stack.
Stack columns created using the COMPUTE command cannot be copied into the Current Area.

If the source (FROM) stack is the Current Area, the only Current Area fields that are copied are those that have a corresponding column name in the target (INTO) stack.

If the target (INTO) stack is not subscripted, the data is copied into the first row in the stack. If the target (INTO) stack is subscripted, the copied row or rows are inserted at this row.

If the COPY command specifies the command output destination as a row or rows of an existing stack that already have data in them, then the old data in these rows is overwritten with the new data when the COPY is executed.

If the source (FROM) stack has fewer columns than the target (INTO) stack, the columns that do not have any data are initialized to blank, zero, or null (missing) as appropriate.

Source (FROM) stack rows will overwrite the specified target (INTO) stack rows if they already exist.

If the COPY command creates rows in the target (INTO) stack, and the target (INTO) stack contains columns that are not in the source (FROM) stack, those columns in the new rows will be initialized to their default values of blank, zero, or null (missing).

If the source (FROM) stack has more columns than the target (INTO) stack, only corresponding columns are copied.

The FOR prefix copies rows from the source (FROM) stack one row at a time, not all at the same time. For example, the following command:

```sql
FOR ALL COPY FROM Car(Car.FocIndex) INTO Car(Car.FocIndex+1);
```

copies the first row into the second, then copies those same values from the second row into the third, and keeps going. When the command has finished executing, all rows will have the same values as the first row.

**Reference: Commands Related to COPY**

- **INFER** defines the columns in a stack.
- **COMPUTE** defines the columns in a stack for non-data source fields.
- **NEXT** defines the columns in a stack and places data into it.
DECLARE

The DECLARE command creates global and local variables (including objects), and gives you the option of assigning an initial value.

Where you place a DECLARE command within a procedure depends on whether you want it to define local or global variables. See Local and Global Declarations on page 86 for more information.

Syntax: How to Use the DECLARE Command

The syntax of the DECLARE command is

```
DECLARE
[(
objectname/datatype [DFC cc YRT yy] [missing]] [= expression];
.
.
.
)]

missing:[MISSING {ON|OFF} [NEEDS] [{SOME|ALL} [DATA]]
```

where:

**objectname**

Is the name of the object or other variable that you are creating. The name is subject to the standard naming rules of the Maintain language. See Specifying Names on page 19 for more information.

**datatype**

Is a data type (a class or built-in format).

**expression**

Is an optional expression that will provide the initial value of the variable. If the expression is omitted, the initial value of the variable is the default for that data type: a space or null for character and date and time data types, and zero or null for numeric data types. When declaring a new variable using a class data type, you must omit expression.

**DFC cc**

Specifies a default century that will be used to interpret any dates with unspecified centuries in expressions assigned to this variable. cc is a two-digit number indicating the century (for example, 19 would indicate the twentieth century). If this is not specified, it defaults to 19.
This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

For information about working with cross-century dates, see *Unique FOCUS Topics* on the WebFOCUS Documentation CD.

**YRT yy**

Specifies a default threshold year for applying the default century identified in DFC cc. *yy* is a two-digit number indicating the year. If this is not specified, it defaults to 00.

When the year of the date being evaluated is less than the threshold year, the century of the date being evaluated defaults to the century defined in DFC cc plus one. When the year is equal to or greater than the threshold year, the century of the date being evaluated defaults to the century defined in DFC cc.

This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

For information about working with cross-century dates, see *Unique FOCUS Topics* on the WebFOCUS Documentation CD.

**missing**

Is used to allow or disallow null values. This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

**MISSING**

If the MISSING syntax is omitted, the default value of the variable is zero for numeric variables and a space for character and date and time variables. If it is included, its default value is null.

**ON**

Sets the default value to null.

**OFF**

Sets the default value to zero or a space.

**NEEDS**

Is an optional keyword that clarifies the meaning of the command for a reader.

**SOME**

Indicates that for the target variable to have a value, some (at least one) of the variables in the expression must have a value. If all of the variables in the expression are null, the target variable will be null. This is the default.
ALL

Indicates that for the target variable to have a value, all the variables in the expression must have values. If any of the variables in the expression is null, the target variable will be null.

DATA

Is an optional keyword that clarifies the meaning of the command for a reader.

( )

Groups a sequence of declarations into a single DECLARE command. The parentheses are required for groups of local declarations, otherwise they are optional.

Reference: Commands Related to DECLARE

- **DESCRIBE** defines classes and data type synonyms.
- **COMPUTE** creates global variables (including objects) and assigns values to existing variables.

Local and Global Declarations

When you declare a new variable, you choose between making the variable:

- **Local** (that is, known only to the function in which it is declared). To declare a local variable, issue the DECLARE command inside the desired function. The DECLARE command must precede all other commands in the function.

  If you wish to declare a local variable in the Top function, note that you cannot issue a DECLARE command in an implied Top function, but you can issue it within an explicit Top function.

- **Global** (that is, known to all the functions in the procedure). To declare a global variable, place the DECLARE command outside of a function (for example, at the beginning of the procedure prior to all functions), or define it using the COMPUTE command anywhere in the procedure. Note that if you place any DECLARE commands at the beginning of the procedure, you must have an explicit TOP case in order to end the global declarations.

We recommend declaring your variables locally, and (when you need to work with a variable outside the function in which it was declared) passing it to the other function as an argument. Local variables are preferable to global variables because they are protected from unintended changes made in other functions.
DELETE

The DELETE command identifies segment instances from a transaction source (a stack or the Current Area) and deletes the corresponding instances from the data source.

When you issue the command, you specify an anchor segment. For each row in the transaction source, DELETE searches the data source for a matching segment instance. When it finds a match, it deletes that anchor instance and all the descendants of the anchor.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This ensures that DELETE can navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the DELETE Command

The syntax of the DELETE command is

\[ \text{[FOR \{int\|ALL\}] \text{ DELETE segment [FROM stack[(row)]] ;} } \]

where:

FOR

Is used with ALL or an integer to specify how many stack rows to use to identify segment instances. If FOR is omitted, one stack row will be used.

When you specify FOR, you must also specify FROM to identify a source stack.

int

Is an integer constant or variable that indicates the number of stack rows to use to identify segment instances to be deleted.

ALL

Specifies that the entire stack is used to delete the corresponding records in the data source.

segment

Specifies the anchor segment of the path you wish to delete. To specify a segment, provide the name of the segment or of a field within the segment.
FROM

Is used to specify a stack whose key columns identify records to delete. If no stack is specified, data from the Current Area is used.

stack

Is a stack name. Only one stack can be specified.

row

Is a subscript that specifies which stack row to begin with.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.

Example: Specifying Which Segments to Delete

The DELETE command removes the lowest specified segment and all of its descendant segments. For example, if a data source structure has four segments in a single path (named First, Second, Third, and Fourth), the command

DELETE First.Field1 Second.Field2;

will delete instances from the Second, Third, and Fourth segments.

If you issue the command

DELETE First.Field1;

you will delete the entire data source path.

Example: Deleting Records Identified in a Stack

In the following example, the data in rows 2, 3, and 4 of the Stkemp stack is used to delete data from the data source. The stack subscript indicates start in the second row of the stack and the FOR 3 means DELETE data in the data source based on the data in the next three rows.

FOR 3 DELETE Emp_ID FROM Stkemp(2);
**Example:**  Deleting a Record Identified in a Form

The first example prompts the user for the employee ID in the EmployeeIDForm form. If the employee is already in the data source, all records for that employee are deleted from the data source. This includes the employee instance in the root segment and all descendent instances (such as pay dates, addresses, and so on). In order to find out if the employee is in the data source, a MATCH command is issued:

```maintain
MAINTAIN FILE Employee
Winform Show EmployeeIDForm;
CASE DELEMP
  MATCH Emp_ID;
  ON MATCH DELETE Emp_ID;
  ON NOMATCH TYPE "Employee id <Emp_ID not found. Reenter";
  COMMIT;
ENDCASE
END
```

When the user presses Enter, function DELEMP is triggered as an event handler from a form. Control is then passed back to EmployeeIDForm.

The second example provides the same functionality. The only difference is that a MATCH is not used to determine whether the employee already exists in the data source. The DELETE can only work if the record exists. Therefore if an employee ID is entered that does not exist, the only action that can be taken is to display a message. In this case, the variable FocError is checked. If FocError is not equal to zero, then the DELETE failed and the message is displayed:

```maintain
MAINTAIN FILE Employee
INFER EMP_ID INTO STACKEMP
Winform Show EmployeeIDForm;
CASE DELEMP
  DELETE Emp_ID;
  IF FocError NE 0 THEN
    TYPE "Employee id <Stackemp.Emp_ID not found. Reenter";
  COMMIT;
ENDCASE
END
```

**Reference:** Usage Notes for DELETE

- Because the DELETE command removes the instance pointed to by the segment position marker, after the deletion the marker has a null value and the segment has no current position. If you need to reestablish position you can issue the REPOSITION command.
You delete a unique segment by deleting its parent. If you wish to erase the fields of a unique segment without affecting its parent, you can instead update its fields to space, zero, or null.

In order for the DELETE to work, the data must exist in the data source. When a set of rows are changed without first finding out if they already exist in the data source, then it is possible that some of the rows in the stack will be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. If you want all rows to be accepted or rejected as a unit, you should treat the stack as a logical transaction, evaluate the FocError transaction variable, and then issue a ROLLBACK command if the entire stack is not accepted. The transaction variable FocErrorRow is automatically set to the number of the first row that failed.

After the DELETE is processed, the transaction variable FocError is given a value. If the DELETE is successful, FocError is zero. If the DELETE fails (for example, the key values do not exist in the data source), FocError is set to a non-zero value and (if the DELETE is set-based) FocErrorRow is set to the number of the row that failed. If there is a concurrency conflict at COMMIT time, the transaction variable FocCurrent is set to a non-zero value.

A DELETE command cannot have more than one input (FROM) stack.

When a DELETE command is complete, the variable FocError is set. If the DELETE is successful (the records to be deleted exist in the data source), then FocError is set to zero. If the records do not exist, FocError is set to a non-zero value. If the DELETE operation was set-based, Maintain sets FocErrorRow to the number of the row that failed.

Maintain requires that the data sources to which it writes have unique keys.

**Reference:** Commands Related to DELETE

- **COMMIT** makes all data source changes since the last COMMIT permanent.
- **ROLLBACK** cancels all data sources changes made since the last COMMIT.

**DESCRIBE**

The DESCRIBE command enables you to define and work with objects (called classes) that cannot be defined using the standard set of classifications. Standard classifications include data types such as Alphanumeric, Numeric, and Integer (a subset of Numeric).

A class can represent a data type synonym. You can assign a name to a specific data type and then use this name as the format specification for variables. In this way, you can change the formats of multiple variables by changing the class definition.
A class can also represent an object consisting of other objects, called components of the class. You can define two types of components for a class, fields and functions (cases).

The DESCRIBE command defines the structure and behavior of a class. You then use the COMPUTE or DECLARE command to create an instance of the class. The COMPUTE command defines a global instance of the class. To create an instance that is local to a specific case, use the DECLARE command within that case.

To reference a component of a class instance, qualify the name of the component with the class instance name, separating them with a period. For example, consider a class called Room, which has components length and width. You can create a class instance named MyRoom in a COMPUTE or DECLARE command. For example:

```plaintext
COMPUTE MyRoom/Room;
```

To reference the length component of the MyRoom instance, qualify the component name with the class instance name:

```plaintext
MyRoom.length
```

This is similar to qualifying a field name in a data source with its file or segment name.

Within the DESCRIBE and ENDDESCRIPTION commands that define a class, you do not qualify component names for that class.

**Syntax:**

**How to Use the DESCRIBE Command**

You must issue the DESCRIBE command outside of a function (case), for example, at the beginning of the procedure prior to all functions.

The syntax of the DESCRIBE command to define a new class is

```plaintext
DESCRIBE classname = ([superclass +] memvar/type [, memvar/type] ...) 
[;]
[memfunction
[memfunction] ...
ENDDESCRIPTION]
```

The syntax of the DESCRIBE command to define a synonym for a data type is

```plaintext
DESCRIBE synonym = datatype ;
```

where:

```plaintext
classname
```

Is the name of the class that you are defining. The name is subject to the standard naming rules of the Maintain language. See *Specifying Names* on page 19 for more information.
**superclass**

Is the name of the superclass from which you wish to derive this class. Used only to describe a subclass.

**memvar**

Names one of the member variables of the class. The name is subject to the standard naming rules of the Maintain language. See *Specifying Names* on page 19 for more information.

**type**

Is a data type (a built-in format or a class).

**memfunction**

Defines one of the member functions of the class. Member functions are defined the same way as other Maintain functions, using the CASE command. See *CASE* on page 69 for more information.

**synonym**

Is a synonym for a data type (a class or format). The synonym is subject to the standard naming rules of the Maintain language. See *Specifying Names* on page 19 for more information.

;  

For class definitions, this terminates the definition if the definition omits member functions. If it includes member functions, the semicolon is omitted and the ENDDESCRIPTIVE command is required.

For synonym definitions, this terminates the definition and is required.

**ENDDESCRIBE**

Ends the class definition if it includes member functions. If it omits member functions, the ENDDESCRIPTIVE command must also be omitted, and the definition terminates with a semicolon.

**Example: Data Type Synonyms**

Data type synonyms can make it easier for you to maintain variable declarations. For example, if your procedure creates many variables for names of people, and defines them all as A30, you would define a data type synonym for A30:

```plaintext
DESCRIBE NameType = A30;
```
You would then define all of the name variables as NameType:

```plaintext
DECLARE UserName/NameType;
COMPUTE ManagerName/NameType;
DECLARE CustomerName/NameType;
```

If you needed to change all name variables to A40, you could change all of them at once simply by changing one data type synonym:

```plaintext
DESCRIBE NameType = A40;
```

**Example:** Defining a Class and Creating an Instance

The following DESCRIBE command defines a class named Room in an architecture application. The components of the class are three fields, Length, Width, and Height:

```plaintext
DESCRIBE Room = (Width/I4, Height/I4, Length/I4);
```

The following COMPUTE command creates an instance of the Room class named abc and assigns values to the components, qualifying each component with the class name:

```plaintext
COMPUTE abc/Room;
abc.Width = 10;
abc.Height = 20;
abc.Length = 30;
```

Once the instance is created, you can use it in other Maintain commands. For example, the following TYPE command types each component value:

```plaintext
TYPE "Width=<abc.Width Height=<abc.Height Length=<abc.Length"
```

**Class Member Functions**

Functions included within a class definition specify operations that can be performed using the components of the class.

Two function names, StartUp and CleanUp, are reserved and have specific uses.

If you define a case called StartUp, that case is executed whenever an instance of the class is created. A global instance is created at the beginning of the Maintain procedure. A local instance is created each time the case in which it is declared is performed.

If you define a case called CleanUp, that case is executed whenever an instance of the class is destroyed (reaches the end of its scope). The scope of a global instance ends after execution of the Maintain procedure. The scope of a local instance ends each time execution returns to the procedure that performed the case in which it was declared.
Reference:  **Startup Case Considerations**

You can create a global instance of a class using the COMPUTE command anywhere in the Maintain procedure. To create an instance local to a specific case, use the DECLARE command within that case.

You can use the Startup case to assign initial values to the components of a global instance of a class.

To pass initial values for class components to the Startup case:

- Define the Startup case to take arguments representing those components (with argument names different from the component names).
- In the Startup case, assign the incoming parameter values to the component field names.
- Then, in the COMPUTE command that creates the instance, specify argument values to pass to case Startup. For example, if the class is named *Room*, the instance is named *MyRoom*, and you want to assign component values length=15 and width= 10, use the following syntax to pass the values to the Startup case:

  \[
  \text{COMPUTE MyRoom/Room(15,10);}
  \]

**Note:** The DECLARE command does not support passing arguments to the Startup case. However, you can always use a COMPUTE or DECLARE command to assign initial or non-initial values:

\[
[\text{COMPUTE|DECLARE}] \text{ MyRoom/Room;}
\text{MyRoom.length = 20;}
\text{MyRoom.width = 15;}
\]

Reference:  **Executing Member Functions**

Just as you can reference components of the class by qualifying the component names with the class instance name, you can execute a function within the class by qualifying the function name with the class instance name. For example, if the *Room* class contains a function called FINDAREA that takes the arguments *length* and *width* and returns the area, you can execute this function and type the returned value with the following commands:

\[
\text{AREA/I4 = MyRoom.FINDAREA(MyRoom.length, MyRoom.width)}
\text{TYPE "AREA = <AREA";}
\]
If the function operates on the components of the class, those components are available to the function without passing them as arguments. Therefore, if \(\text{length}\) and \(\text{width}\) are components of the \textit{Room} class, the \text{FINDAREA} function does not need to take any arguments. In this case, you invoke the function as follows:

\[
\text{AREA/I4} = \text{MyRoom.FINDAREA()}
\]

\[
\text{TYPE } \text{"AREA} = <\text{AREA}\";
\]

Note that parentheses are required when invoking a member function even if the function does not take arguments.

**Example: Defining a Class**

The \text{DESCRIBE} command in the following Maintain procedure defines a class named \textit{Floor} in an architecture application. The components of the class are three fields (\text{Length}, \text{Width}, and \text{Area}) and one case (\text{PrintFloor}). The \text{COMPUTE} command creates an instance of the class named \text{MYFLOOR}, assigns values to the components, and calls the \text{PrintFloor} function. Although the \text{PrintFloor} function does not take arguments, the parentheses are needed to identify \text{PrintFloor} as a function:

```
\text{MAINTAIN}  
\text{DESCRIBE Floor} = (\text{Length/I4, Width/I4, Area/I4})  
\text{CASE PrintFloor}  
\text{\hspace{1cm} TYPE } \text{"length}=<\text{Length } \text{width}=<\text{Width } \text{area}=<\text{Area}\";  
\text{\hspace{1cm} ENDCASE}  
\text{ENDDESCRIBE}  
\text{COMPUTE MYFLOOR/FLOOR;}  
\text{MYFLOOR.Length} = 15;  
\text{MYFLOOR.Width} = 10;  
\text{MYFLOOR.Area} = \text{MYFLOOR.Length} \times \text{MYFLOOR.Width};  
\text{MYFLOOR.PrintFloor();}  
\text{END}
```

The output is:

```
length=15 width=10 area=150
```
### Example: Defining a Class With a Startup Case

The DESCRIBE command in the following Maintain procedure defines a class named Floor in an architecture application. The components of the class are three fields (Length, Width, and Area) and one case (PrintFloor). The COMPUTE command creates an instance of the class named MYFLOOR, passes values for the components to the Startup case, and calls the PrintFloor function. The Startup case initializes the component fields with the values passed in the COMPUTE command:

```maintain
define class Floor
  (Length/I4, Width/I4, Area/I4)
end define

maintain
case startup takes L/I4, W/I4, A/I4
  length = L;
  width = W;
  area = A;
endcode

case printfloor
type "In PrintFloor: length=<Length  width=<Width  area=<Area";
endcode
end

compute myfloor/floor(15, 10, 150);
type "After Startup: LENGTH=<myfloor.length  WIDTH=<myfloor.width  AREA=<myfloor.area";
myfloor.printfloor();
end
```

The output is:

```
After Startup: LENGTH=15  WIDTH=10  AREA=150
In PrintFloor: length=15 width=10 area=150
```

### Example: Defining and Using a Local Class Instance

In the following Maintain procedure, the DESCRIBE command defines a class named FNAME with components LAST and FIRST. The FORMNAME case concatenates the last and first names and separates them with a comma and a space.

The main procedure loops through the first five records of the EMPLOYEE data source, and passes each last name and first name to case PRTFULL.

Case PRTFULL creates a local instance of the FNAME class, invokes the FORMNAME member function, and types the full name returned from that class. Although FORMNAME does not take any arguments, the parentheses used when invoking FORMNAME identify it as a function.
TYPE commands in each case illustrate the flow of control:

MAINTAIN FILE VIDEOTRK
DESCRIBE FNAME = (LAST/A15, FIRST/A10)

CASE STARTUP;
TYPE "IN CASE STARTUP: I = <I";
ENDCASE

CASE FORMNAME RETURNS FULLNAME/A30;
FULLNAME/A30 = LAST || ', ' || FIRST;
TYPE "IN CASE FORMNAME: I = <I  FULLNAME = <FULLNAME";
ENDCASE

CASE CLEANUP;
TYPE "IN CASE CLEANUP: I = <I";
ENDCASE
ENDDESCRIBE

-* MAIN PROCEDURE

FOR 5 NEXT CUSTID INTO CUSTSTK;
REPEAT 5 I/I1 = 1;
COMPUTE LAST/A15 = CUSTSTK(I).LASTNAME;
COMPUTE FIRST/A10 = CUSTSTK(I).FIRSTNAME;
TYPE "IN MAIN PROCEDURE: I = <I  LAST = <LAST  FIRST = <FIRST";
PERFORM PRTFULL(LAST, FIRST);
ENDREPEAT I = I+1;

CASE PRTFULL TAKES LAST/A15, FIRST/A10;
-* MEMNAME IS A LOCAL VARIABLE
DECLARE MEMNAME/FNAME;
MEMNAME.LAST=LAST;
MEMNAME.FIRST=FIRST;
NEWNAME/A30 = MEMNAME.FORMNAME();
TYPE "IN CASE PRTFULL: I = <I MEMBER NAME IS <NEWNAME";
ENDCASE
END
The output shows that the Startup case is called prior to each invocation of case PRTFULL (which defines the local instance), and the Cleanup case is called at the end of each invocation of case PRTFULL:

IN MAIN PROCEDURE: I = 1 LAST = CRUZ FIRST = IVY
IN CASE STARTUP: I = 1
IN CASE FORMNAME: I = 1 FULLNAME = CRUZ, IVY
IN CASE PRTFULL: I = 1 MEMBER NAME IS CRUZ, IVY
IN CASE CLEANUP: I = 1
IN MAIN PROCEDURE: I = 2 LAST = HANDLER FIRST = EVAN
IN CASE STARTUP: I = 2
IN CASE FORMNAME: I = 2 FULLNAME = HANDLER, EVAN
IN CASE PRTFULL: I = 2 MEMBER NAME IS HANDLER, EVAN
IN CASE CLEANUP: I = 2
IN MAIN PROCEDURE: I = 3 LAST = WILSON FIRST = KELLY
IN CASE STARTUP: I = 3
IN CASE FORMNAME: I = 3 FULLNAME = WILSON, KELLY
IN CASE PRTFULL: I = 3 MEMBER NAME IS WILSON, KELLY
IN CASE CLEANUP: I = 3
IN MAIN PROCEDURE: I = 4 LAST = KRAMER FIRST = CHERYL
IN CASE STARTUP: I = 4
IN CASE FORMNAME: I = 4 FULLNAME = KRAMER, CHERYL
IN CASE PRTFULL: I = 4 MEMBER NAME IS KRAMER, CHERYL
IN CASE CLEANUP: I = 4
IN MAIN PROCEDURE: I = 5 LAST = GOODMAN FIRST = JOHN
IN CASE STARTUP: I = 5
IN CASE FORMNAME: I = 5 FULLNAME = GOODMAN, JOHN
IN CASE PRTFULL: I = 5 MEMBER NAME IS GOODMAN, JOHN
IN CASE CLEANUP: I = 5

TRANSACTIONS: COMMITS = 1 ROLLBACKS = 0
SEGMENTS : INCLUDED = 0 UPDATED = 0 DELETED = 0

**Example:** **Defining and Using a Global Class Instance**

In the following Maintain procedure, the DESCRIBE command defines a class named FNAME with components LAST and FIRST. The FORMNAME case concatenates the last and first names and separates them with a comma and a space.

The main procedure loops through the first five records of the EMPLOYEE data source, and passes each last name and first name to case PRTFULL.

Case PRTFULL creates a global instance of the FNAME class, invokes the FORMNAME member function, and types the full name returned from that class. Although FORMNAME does not take any arguments, the parentheses used when invoking FORMNAME identify it as a function.
TYPE commands in each case illustrate the flow of control:

```
MAINTAIN FILE VIDEOTRK
DESCRIBE FNAME = (LAST/A15, FIRST/A10)
  CASE STARTUP TAKES LASTNAME/A15, FIRSTNAME/A10;
    TYPE "IN CASE STARTUP: I = <I  LAST = <LASTNAME  FIRST = <FIRSTNAME";
      LAST = LASTNAME;
      FIRST = FIRSTNAME;
    ENDCASE
  CASE FORMNAME RETURNS FULLNAME/A30;
    FULLNAME/A30 = LAST || ' ', ' ' || FIRST;
    TYPE "IN CASE FORMNAME: I = <I  FULLNAME = <FULLNAME";
  ENDCASE
  CASE CLEANUP;
    TYPE "IN CASE CLEANUP: I = <I ";
  ENDCASE
ENDDESCRIBE

*-MAIN PROCEDURE

FOR 5 NEXT CUSTID INTO CUSTSTK;

REPEAT 5 I/I1 = 1;
  COMPUTE LAST/A15 = CUSTSTK(I).LASTNAME;
  COMPUTE FIRST/A10 = CUSTSTK(I).FIRSTNAME;
  TYPE "IN MAIN PROCEDURE: I = <I  LAST = <LAST  FIRST = <FIRST";
  PERFORM PRTFULL(LAST, FIRST);
ENDREPEAT I = I+1;

CASE PRTFULL TAKES LAST/A15, FIRST/A10;
  COMPUTE MEMNAME/FNAME('ABEL', 'AARON');
  NEWNAME/A30  = MEMNAME.FORMNAME();
  TYPE "IN CASE PRTFULL: I = <I  MEMBER NAME IS <NEWNAME";
ENDCASE
END
```
The output shows that the Startup case is called at the start of the Maintain procedure, and the Cleanup case is called following the execution of the entire Maintain procedure:

```
IN CASE STARTUP: I = 0 LAST = ABEL FIRST = AARON

IN MAIN PROCEDURE: I = 1 LAST = CRUZ FIRST = IVY
IN CASE FORMNAME: I = 1 FULLNAME = CRUZ, IVY
IN CASE PRTFULL: I = 1 MEMBER NAME IS CRUZ, IVY
IN MAIN PROCEDURE: I = 2 LAST = HANDLER FIRST = EVAN
IN CASE FORMNAME: I = 2 FULLNAME = HANDLER, EVAN
IN CASE PRTFULL: I = 2 MEMBER NAME IS HANDLER, EVAN
IN MAIN PROCEDURE: I = 3 LAST = WILSON FIRST = KELLY
IN CASE FORMNAME: I = 3 FULLNAME = WILSON, KELLY
IN CASE PRTFULL: I = 3 MEMBER NAME IS WILSON, KELLY
IN MAIN PROCEDURE: I = 4 LAST = KRAMER FIRST = CHERYL
IN CASE FORMNAME: I = 4 FULLNAME = KRAMER, CHERYL
IN CASE PRTFULL: I = 4 MEMBER NAME IS KRAMER, CHERYL
IN MAIN PROCEDURE: I = 5 LAST = GOODMAN FIRST = JOHN
IN CASE FORMNAME: I = 5 FULLNAME = GOODMAN, JOHN
IN CASE PRTFULL: I = 5 MEMBER NAME IS GOODMAN, JOHN
```

**Transactions:**
- Commits: 1
- Rollbacks: 0
- Segments: Included: 0, Updated: 0, Deleted: 0

```
IN CASE CLEANUP: I = 6
```

**Defining and Using SuperClasses and Subclasses**

After you describe a class, you can derive other classes from it. Subclasses inherit member variables and functions from their superclasses.

Order of classes matters when defining superclasses and subclasses. You must describe a superclass prior to its subclasses.

A class can also use another class as one of its components. Again, order matters. You must describe the class you are using as a component prior to the class that uses it.

**Example: Defining a Subclass**

The following example describes two classes, Floor and Room. Floor consists of components Length and Width and member function FloorArea.

Room is a subclass of Floor. It inherits components Length and Width and member function FloorArea. It adds component Depth and function RoomVolume.
The main procedure creates an instance of Room called MYROOM. It then assigns values to the components, including inherited components Length and Width. It invokes the inherited member function FloorArea as well as the RoomVolume function.

The main procedure then types the component values and the values returned by the member functions.

**MAINTAIN**

```maintain
DESCRIBE Floor = (Length/I4, Width/I4)
    CASE FloorArea RETURNS Area/I4;
        Area = Length * Width;
    ENDCASE
ENDDESCRIBE

DESCRIBE Room = (Floor + Depth/I4)
    CASE RoomVolume RETURNS Volume/I4;
        Volume = FLOORAREA() * Depth;
    ENDCASE
ENDDESCRIBE

COMPUTE MYROOM/ROOM;
MYROOM.LENGTH = 15;
MYROOM.WIDTH = 10;
MYROOM.DEPTH = 10;
AREA/I4 = MYROOM.FLOORAREA();
VOLUME/I4 = MYROOM.RoomVolume();
TYPE "LENGTH=<MYROOM.Length , WIDTH=<MYROOM.Width , " |
    "DEPTH=<MYROOM.DEPTH , AREA = <AREA , " |
    "VOLUME=<VOLUME";
END
```

The output is:

```
LENGTH=15, WIDTH=10, DEPTH=10, AREA = 150, VOLUME=1500
```

**Example: Using a Class as a Component of Another Class**

The following example describes three classes: RoomDetail, Floor, and Room.

RoomDetail has no member functions. It consists of two components, Depth and RmType.

Floor consists of components Length and Width and member function FloorArea.
Room is a subclass of Floor. It inherits components Length and Width and member function FloorArea. In addition, it has member function RoomVolume and component RmType, which is an instance of class RoomDetail. When referring to the components of RoomDetail, you must qualify them with their instance name. For example, the Rtype component is referenced as follows:

```
RmType.Rtype
```

The main procedure creates an instance of Room called MYROOM. It then assigns values to the components, including inherited components Length and Width. When assigning values to the components of the RmType instance of the RoomDetail class, it must qualify them with the instance name, MYROOM. Since these names already had one level of qualification when they were referenced in the Room class, they now have two levels of qualification. For example, the following assigns the value 10 to the Depth component:

```
MYROOM.RmType.Depth = 10;
```

Note that when no ambiguity in the variable name will result, only one level of qualification is actually enforced. MYROOM.Depth is understood as MYROOM.RmType.Depth since Depth does not appear in any other context.
The main procedure invokes the inherited member function FloorArea as well as the RoomVolume function, then types the component values and the values returned by the member functions.

```
MAINTAIN
    DESCRIBE RoomDetail = (Depth/I4, Rtype/A10);
    DESCRIBE Floor = (Length/I4, Width/I4)
        CASE FloorArea  RETURNS Area/I4;
            Area = Length * Width;
        END_CASE
    END_DESCRIBE
    DESCRIBE Room  = (Floor + RmType/RoomDetail)
        CASE RoomVolume RETURNS Volume/I4;
            Volume = FLOORAREA() * RmType.Depth;
            TYPE "ROOM TYPE IS <RmType.Rtype";
        END_CASE
    END_DESCRIBE
    COMPUTE MYROOM/ROOM;
    MYROOM.LENGTH = 15;
    MYROOM.WIDTH = 10;
    MYROOM.RmType.Depth = 10;
    MYROOM.RmType.Rtype = 'DINING    ';
    AREA/I4 = MYROOM.FLOORAREA();
    VOLUME/I4 = MYROOM.RoomVolume();
    TYPE "LENGTH=<MYROOM.LENGTH , WIDTH=<MYROOM.WIDTH , "|
    "DEPTH=<MYROOM.rmtype.DEPTH , AREA = <AREA ," | 
        " VOLUME=<VOLUME";
END
```

The output is:

```
ROOM TYPE IS DINING
LENGTH=15, WIDTH=10, DEPTH=10, AREA = 150, VOLUME=1500
```

**Reference:** Commands Related to DESCRIBE

- **DECLARE** creates local and global variables, including objects.
- **COMPUTE** creates global variables, including global objects, and assigns values to existing variables.
ENCRYPT/DECRYPT

These commands are outside the Maintain language, but are described in this chapter for your convenience. You can issue these commands from within a WebFOCUS procedure, not from within a Maintain procedure.

Since the restriction information for a FOCUS data source is stored in its Master File, you might want to encrypt the Master File in order to prevent users from examining the restriction rules. Only the database administrator can encrypt a Master File. You must set PASS=DBA name before you issue the ENCRYPT command.

The following is an example of the complete procedure:

```plaintext
SET PASS=JONES76
ENCRYPT FILE PERS
```

The process can be reversed if you wish to change the restrictions. The command to restore the Master File to a readable form is DECRYPT.

The DBA password must be issued with the SET command before the Master File can be decrypted.

For example:

```plaintext
SET PASS=JONES76
DECRYPT FILE PERS
```

Encrypting Procedures

Once PASS is set, it is also possible to encrypt procedures by using the same ENCRYPT syntax. If no file extension is specified, the ENCRYPT command assumes an extension of .MAS, so be sure to include the extension when encrypting procedures. Also, remember to include the whole name of the file being encrypted, with no wildcards.

When encrypting any file, it is very important to enter its whole name after the ENCRYPT command. The use of wildcards could damage all the files that fit the file specification with the wildcards, and is therefore very dangerous.

Encrypting Data

You can also use the ENCRYPT command within the Master File to encrypt some or all of the data.
Encryption takes place on the segment level (that is, the entire segment is encrypted). The request for encryption is made in the Master File by setting the attribute ENCRYPT to ON. For instance:

```
SEGMENT=COMPSEG, PARENT=IDSEG, SECTYPE=S1, ENCRYPT=ON,$
```

You must specify the ENCRYPT attribute when the file is new, before it contains any data. Encryption cannot be requested later by a change to the Master File and cannot be removed once it has been requested and any data has been entered in the file.

**Note:** Encryption is used only with FOCUS data sources.

### Performance Considerations

There is a small loss in processing efficiency when data is encrypted. You can minimize this loss by grouping the sensitive data fields and making them a separate segment with SECTYPE=U beneath their original segment.

For example, suppose the data items on a segment are:

![Diagram of data items and sensitive field](image)

They should be grouped as:

![Diagram of grouped data items](image)
Restricting Existing Data Sources

When you write a Master File for a new data source and include security limitations, data added to the data source is automatically protected according to those rules. If you write a new Master File for an existing data source that contains no data, that data will also be automatically protected. If, however, you have existing FOCUS data sources to which you want to add security limitations, you need to use the RESTRICT command.

**Note:** This is not the RESTRICT attribute.

The syntax is

```
RESTRICT C:filename.FOC
END
```

where:

`filename`

Is the name of the data file that you want to protect.

Remember to specify the disk drive letter in this command. If you omit it, you will receive an error message. Also remember to include the file extension. Without it, the RESTRICT command will not be able to find the file.

RESTRICT is actually the next to last of a series of steps that are necessary when changing or adding a password to your data. The following outlines the steps:

1. Edit the Master File, changing the DBA attribute to equal the new password.
2. Issue the command `SET PASS= the new password`.
3. Issue the CHECK FILE command to load your edited Master File into memory.
4. The FOCUS data file still has the old password stored in it, so issue `SET PASS= the old password` so that you have the right to use the RESTRICT command on the data file. (If you are adding a password to a file that has none, this step is unnecessary.)
5. Issue the RESTRICT command with the FOCUS data file as the parameter.
6. `SET PASS= the new password`.

The file now has the new password assigned to it.

**Note:** Before you begin this procedure, make sure to back up your data file.

END

The END command marks the end of a Maintain procedure and terminates its execution.
Syntax: How to Use the END Command

The syntax of the END command is

```
END
```

where:

```
END
```

Is the last line of the procedure, and must be coded in uppercase letters.

Reference: Commands Related to END

- **MAINTAIN** is used to initiate the parsing and execution of a Maintain procedure.
- **CALL** is used to call one procedure from another.

EXEC

The EXEC command enables you to call a WebFOCUS procedure and pass parameters to and from the procedure. You can run any WebFOCUS procedure residing on a WebFOCUS Server accessible to the WebFOCUS Server where the calling procedure resides. From a WebFOCUS procedure you can run many other types of procedures, including compiled C programs, CICS transactions, and native RDBMS command files.

For more information, see the *Developing WebFOCUS Maintain Applications* manual.

Syntax: How to Use the EXEC Command

The syntax of the EXEC command is

```
EXEC progname AT server [KEEP|DROP] [PATH {VAR|LIST}] [FROM var_list] [INTO stacks] [;]
```

where:

```
progname
```

Is the name of the external procedure residing on the remote WebFOCUS Server.

```
AT server
```

Identifies the server of the EXEC procedure.

In most situations the Maintain Development Environment supplies this phrase during the deployment phase and you should refrain from coding it yourself. If you do hard code the AT server clause in your procedure, it will not be changed during deployment. A variable can be used to supply the server name.
**PATH**

Is used to specify additional locations (search paths) the system should use when searching for dependent resources (Master Files, imported modules, and others). The path location names are application names existing within the APPROOT directory structure or application names that have been introduced with the APP MAP command. The search path value can be in the form of a Maintain variable or a list of literal values enclosed in quotes, as follows:

```plaintext
EXEC Procedure AT server PATH "AppDir1 AppDir2 AppDir3" ;
EXEC Procedure AT server PATH MyVariable ;
```

**FROM**

Is included to pass one or more variables to the external procedure.

**INTO**

Is included to identify the data stack to receive the answer set or sets coming from the external procedure.

**var_list**

Is one or more scalar variables that you pass to the target procedure, where they are received as numbered amper variables. You can pass any scalar variable except for those defined as variable-length character variables (that is, except for those defined as A0 or TX). Unlike the CALL command, you cannot pass stacks to the target procedure.

The length of a single parameter cannot exceed 32,000 characters. The total length of all specified parameters cannot exceed 32,000 characters.

**stacks**

Is one or more stacks, each of which will receive an answer set from the target procedure. To retrieve multiple answer sets, specify multiple stacks. The stacks are populated sequentially as each answer set is returned by the external procedure. You can pass any stack except for those defined using STACK OF.

The number of variables specified in the EXEC command must not exceed the number returned by the external procedure. If the number specified is fewer than the number returned, the extra returned parameters are ignored.

**KEEP | DROP**

The DROP parameter terminates the server session that the AT server phrase creates if the called procedure is on a different server. The KEEP parameter leaves the server session active for reuse by subsequent external procedures. KEEP is the default.
Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the semicolon, see Terminating Command Syntax on page 24.

**FocCount**

The FocCount stack variable contains the number of rows in the stack. In an empty stack, FocCount is 0. This variable is automatically maintained and the user does not need to do anything when new rows are added or deleted from the stack. For example, the following stack variable contains the number of rows in the EmpInfo stack:

```
EmpInfo.FocCount
```

The FocCount variable is useful as a test to see whether a data source retrieval command is successful. For example, after putting data into a stack, FocCount can be checked to see if its value is greater than zero. FocCount can also be used to perform an action on every row in a stack. A repeat loop can be set up to loop the number of times specified by the FocCount variable.

The following example computes a new salary for each row retrieved from the data source:

```
FOR ALL NEXT Emp_ID Curr_Sal INTO Pay;
COMPUTE Pay.NewSal/D12.2=;
REPEAT Pay.FocCount Cnt/I4=1;
    COMPUTE Pay(Cnt).NewSal/D12.2 = Pay(Cnt).Curr_Sal * 1.05;
ENDREPEAT Cnt=Cnt+1;
```

**FocCurrent**

FocCurrent contains the return code from logical transaction processing. This variable indicates whether or not there is a conflict with another transaction. If the variable value is zero, there is no conflict and the transaction is accepted. If the value is not zero, there is a conflict. FocCurrent is set after each COMMIT and ROLLBACK command.

FocCurrent is local to a procedure. If you wish a given FocCurrent value to be available to another procedure, you must pass it to that procedure as an argument.

**FocError**

FocError contains the return code from the INCLUDE, UPDATE, and DELETE commands. If all the rows in the stack are successfully processed, FocError is set to zero. FocError is set to a non-zero value if:

- INCLUDE rejects the input.
- UPDATE rejects the update.
- DELETE rejects the delete.
- REVISE rejects the changes.

FocError is a global variable. You do not need to pass it between procedures. Its value is cleared each time a Maintain procedure is called.

**FocErrorRow**

After any set-based data source operation (FOR ... UPDATE, DELETE, REVISE, or INCLUDE), if FocError is set to a non-zero value, then FocErrorRow is the number of the row that caused the error.

FocErrorRow is local to a procedure. If you wish a given FocErrorRow value to be available to another procedure, you must pass it to that procedure as an argument.

**FocFetch**

FocFetch contains the return code of the most recently issued NEXT or MATCH command. If the NEXT or MATCH command returned data, FocFetch is set to zero. Otherwise, it is set to a non-zero value.

It is recommended that you test FocFetch in place of issuing the ON NEXT, ON NONEXT, ON MATCH, and ON NOMATCH commands: FocFetch accomplishes the same thing more efficiently.

For example:

```plaintext
FOR ALL NEXT CustID INTO CustOrderStack;  
IF FocFetch NE 0 THEN ReadFailed();  

FocFetch is local to a procedure. If you wish a given FocFetch value to be available to another procedure, you must pass it to that procedure as an argument.

**FocIndex**

The FocIndex stack variable is a pointer to the current instance in a stack. In an empty stack, FocIndex is 1.

This variable is manipulated by the developer and can be used to do things such as determine which row of a stack is to be displayed on a form. A form displays data from a stack based on the value of FocIndex. For example, if a form currently displays data from the PayInfo stack and the following compute is issued:

```plaintext
COMPUTE PayInfo.FocIndex=15;
```
FocMsg

FocMsg is a system stack with one A80 column named Msg. When a Maintain procedure executes either an external procedure or a Maintain procedure on a remote server (that is, a Maintain procedure called using the CALL procname AT command), all of the messages that the called procedure writes to the default output device are automatically copied to the FocMsg stack of the calling procedure. This includes messages issued by TYPE and SAY commands that do not specify a file, and informational and error messages.

If the external procedure calls other external procedures, all messages posted by the chain of external procedures are copied to the same FocMsg stack in the calling Maintain procedure. Non-WebFOCUS logic (such as a compiled 3GL program or a CICS transaction) that is called from an external procedure does not copy to FocMsg.

FocMsg is global to each Maintain procedure.

**Example:**  Cycling Through All the Messages in FocMsg

You can use FocCount to cycle through all of the messages that have been posted to FocMsg:

```plaintext
COMPUTE Counter/I3=1;
REPEAT FocMsg.FocCount;
    TYPE "<FocMsg(Counter).Msg";
    COMPUTE Counter=Counter+1;
ENDREPEAT
```

**Example:**  Retrieving Messages Posted by an External Procedure

This example illustrates how to retrieve messages that were posted by an external procedure.
Client Procedure

1. MAINTAIN FILE MOVIES
2. INFER MovieCode Title INTO MoviesInfo;
3. EXEC GetMovie AT ReprtSrv INTO MoviesInfo;
4. COMPUTE I/I4=1;
5. REPEAT 3;
6. TYPE
7. "Movie code is: << MoviesInfo(I).MovieCode"
8. "    Title: << MoviesInfo(I).Title";
9. COMPUTE I=I+1;
10. ENDBEGIN
11.
12. COMPUTE I=1;
13. REPEAT FocMsg.FocCount;
14. TYPE "Here are the messages from the server: <<FocMsg(I).Msg";
15. COMPUTE I=I+1;
16. ENDBEGIN
17. END

External procedure GetMovie

1. TABLE FILE MOVIES
2. PRINT MOVIECODE TITLE
3. ON TABLE PCHOLD
4. END
5. RUN
6. -TYPE "Finished with the movies retrieval"

GOTO

The GOTO command is used to transfer control to a different Maintain function, to a special point within the current function, or to terminate the application.

If you wish to transfer control to a different function, it is recommended that you use the PERFORM command instead of GOTO.

Syntax: How to Use the GOTO Command

The syntax of the GOTO command is

GOTO destination [;]

where:

destination

Is one of the following:
functionname specifies the name of the function that control is transferred to. Maintain expects to find a function by that name in the procedure. You cannot use GOTO with a function that has parameters.

Top transfers control to the beginning of the Top function. All local variables are freed; current data source positions are retained, as are any uncommitted data source transactions.

END [KEEP | RESET]
Terminates the procedure. Control returns to whatever called the procedure. No function may be named END, as such a function would be ignored and never executed.

KEEP terminates a called procedure, but keeps its data (the values of its variables and data source position pointers) in memory. It remains in memory through the next call or, if it is not called again, until the application terminates.

RESET terminates a called procedure and clears its data from memory. This is the default.

EXIT
This is similar to GOTO END but immediately terminates all procedures in an application. This means that if one procedure calls another and the called procedure issues a GOTO EXIT, both procedures are ended by the GOTO EXIT command. No function may be named EXIT.

ENDCASE
Transfers control to the ENDCASE command in the function, and the function is exited. For information about the ENDCASE command, see CASE on page 69.

ENDREPEAT
Transfers control to the ENDREPEAT command in the current REPEAT loop. The loop is not exited. All appropriate loop counters specified on the ENDREPEAT command are incremented. For information about the REPEAT and ENDREPEAT commands, see REPEAT on page 162.

EXITREPEAT
Exits the current REPEAT loop. Control transfers to the next line after the ENDREPEAT command. For information about the REPEAT and ENDREPEAT commands, see REPEAT on page 162.

;
Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.
For example, to branch to the function named MainMenu, you would issue the command:

GOTO MainMenu

**Reference:** Usage Notes for GOTO

- If the GOTO specifies a function name that does not exist in the program, an error occurs at parse time, which occurs before execution.

- When one procedure calls another, and the called procedure has a GOTO END command, GOTO END ends only the called procedure. The calling procedure is unaffected. A GOTO END does not cause a COMMIT. This allows a called procedure to exit and have the calling program issue the COMMIT when appropriate. For information about the COMMIT command, see **COMMIT** on page 73.

**Reference:** Commands Related to GOTO

- **PERFORM** control to another function. When the function finishes, control is returned to the command following the PERFORM.

- **CASE /ENDCASE** allows a set of commands to be grouped together.

- **REPEAT /ENDBACKUP** provides a general looping facility.

**Using GOTO With Data Source Commands**

A GOTO command can be executed in a MATCH command following an ON MATCH or ON NOMATCH command, or in NEXT following ON NEXT or ON NONEXT. The following syntax branches to the function MatchEdit when a MATCH occurs:

ON MATCH GOTO MatchEdit;

**GOTO and ENDCASE**

When control is transferred to a function with the GOTO command, every condition for exiting that function must contain a command indicating where control should be passed to. If an ENDCASE command is reached by either GOTO or normal program flow, and Maintain has not received any instructions as to where to go next, Maintain takes a default action and exits the procedure. ENDCASE is treated differently when GOTO and PERFORM are combined. See **PERFORM** on page 160 for more information.

**GOTO and PERFORM**

It is recommended that you do not issue a GOTO command within the scope of a PERFORM command.
The scope of a PERFORM command extends from the moment at which it is issued to the moment at which control returns normally to the command or form control point immediately following it. The scope includes any additional PERFORM commands nested within it.

For example, if the Top function issues a PERFORM command to call Case One, Case One issues a PERFORM command to call Case Two. Case Two issues a PERFORM command to call Case Three, and control then returns to Case Two, returns from there to Case One, and finally returns to the Top function. You should not issue a GOTO command from the time the original PERFORM branches out of the Top function until it returns to the Top function.

If, when you code your application, you cannot know every potential run time combination of PERFORM and GOTO branches, it is recommended that you refrain from coding any GOTO commands in your application.

**IF**

The IF command allows conditional processing depending on how an expression is evaluated.

**Syntax: How to Use the IF Command**

The syntax of the IF command is

```
IF boolean_expr THEN maint_command [ELSE maint_command]
```

where:

- `boolean_expr` is an expression that resolves to a value of true (1) or false (0), and can include stack cells and user-defined fields. For more information about Boolean expressions, see *Expressions Reference* on page 27.

Maintain handles the format conversion in cases where the expressions have a format mismatch. If the conversion is not possible, an error message is displayed. For additional information, see *Expressions Reference* on page 27.
It is highly recommended that parentheses be used when combining expressions. If parentheses are not used, the operators are evaluated in the following order:

1. **
2. * /
3. + –
4. LT LE GT GE
5. EQ NE
6. OMITS CONTAINS
7. AND
8. OR

You can place any Maintain command inside an IF command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE.

**Example: Simple Conditional Branching**

The following uses an IF command to compare variable values. The function No_ID is performed if the Current Area value of Emp_ID does not equal the value of Emp_ID in Stackemp:

```
IF Emp_ID NE Stackemp(StackEmp.FocIndex).Emp_ID THEN PERFORM No_ID;
ELSE PERFORM Yes_ID;
```

You might also use an IF command to issue another Maintain command. This example causes a COMMIT if there are no errors:

```
IF FocCurrent EQ 0 THEN COMMIT;
```

**Example: Using BEGIN to Execute a Block of Conditional Code**

This example executes a set of code depending on the value of Department. Additional IF commands could be placed within the BEGIN block of code:

```
IF Department EQ 'MIS' THEN BEGIN
  .
  .
ENDBEGIN
ELSE IF Department EQ 'MARKETING' THEN BEGIN
  .
  .
```
**Example: Nesting IF Commands**

IF commands can be nested as deeply as needed, allowing only for memory constraints. The following shows an IF command nested two levels. There is only one IF command after each ELSE:

```plaintext
IF Dept EQ 1 THEN TYPE "DEPT EQ 1";
ELSE IF Dept EQ 2 THEN TYPE "DEPT EQ 2";
ELSE IF Dept EQ 3 THEN TYPE "DEPT EQ 3";
ELSE IF Dept EQ 4 THEN TYPE "DEPT EQ 4";
```

This example can be executed more efficiently by issuing the following command:

```plaintext
TYPE "DEPT EQ <Dept";
```

You can also use the BEGIN command to place another IF within a THEN phrase. For example:

```plaintext
IF A EQ 1 THEN BEGIN
  IF B EQ 1 THEN BEGIN
    IF C EQ 1 THEN PERFORM C111;
    IF C EQ 2 THEN PERFORM C112;
    IF C EQ 3 THEN PERFORM C113;
    ENDBEGIN
  ELSE IF B EQ 2 THEN BEGIN
    IF C EQ 1 THEN PERFORM C121;
    IF C EQ 2 THEN PERFORM C122;
    IF C EQ 3 THEN PERFORM C123;
    ENDBEGIN
  ENDBEGIN
ELSE IF A EQ 2 THEN BEGIN
  IF B EQ 1 THEN BEGIN
    IF C EQ 1 THEN PERFORM C211;
    IF C EQ 2 THEN PERFORM C212;
    IF C EQ 3 THEN PERFORM C213;
    ENDBEGIN
  ELSE IF B EQ 2 THEN BEGIN
    IF C EQ 1 THEN PERFORM C221;
    IF C EQ 2 THEN PERFORM C222;
    IF C EQ 3 THEN PERFORM C223;
    ENDBEGIN
  ENDBEGIN
ELSE TYPE "A, B AND C did not have expected values";
```
Coding Conditional COMPUTE Commands

When you need to assign a value to a variable, and the value you assign is conditional upon the truth of an expression, you can use a conditional COMPUTE command. Maintain offers you two methods of coding this, using either:

- **An IF command** with two COMPUTE commands embedded within it. For example:

  ```
  IF Amount GT 100
  THEN COMPUTE Tfactor/I6 = Amount;
  ELSE COMPUTE Tfactor = Amount * (Factor - Price) / Price;
  ```

- **A conditional expression** within a COMPUTE command. For example:

  ```
  COMPUTE Tfactor/I6 = IF Amount GT 100 THEN Amount
  ELSE Amount * (Factor - Price) / Price;
  ```

The two methods are equivalent.

**INCLUDE**

The INCLUDE command inserts segment instances from a transaction source (a stack or the Current Area) into a data source.

When you issue the command, you specify a path running from an anchor segment to a target segment. For each row in the transaction source, INCLUDE searches the data source for matching segment instances and, if none exist, writes the new instances from the transaction source to the data source.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This ensures that INCLUDE can navigate from the root to the first instance of the anchor segment.

**Syntax:** How to Use the INCLUDE Command

The syntax of the INCLUDE command is

```
[FOR {int|ALL}] INCLUDE path_spec [FROM stack[(row)]] [;]
```

where:

- **FOR**
  
  Is used with ALL or an integer to specify how many stack rows to add to the data source. If FOR is omitted, one stack row will be added.

  When you specify FOR, you must also specify FROM to identify a source stack.
**int**

Is an integer constant or variable that indicates the number of stack rows to add to the data source.

**ALL**

Specifies that the entire stack is to be added to the data source.

**path_spec**

Identifies the path to be added to the data source. To identify a path, specify its anchor and target segments. (You cannot specify a unique segment as the anchor.) If the path contains only one segment, and the anchor and target are identical, simply specify the segment once. For paths with multiple segments, if you wish to make the source code clearer to readers, you can also specify segments between the anchor and target.

To add a unique segment instance to a data source, you must explicitly specify the segment in `path_spec`. Otherwise, the unique segment instance will not be added even if it is on the path between the anchor and target segments. This preserves the advantage of assigning space for a unique segment instance only when the instance is needed.

To specify a segment, provide the name of the segment or of a field within the segment.

**FROM**

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

**stack**

Is a stack name. Only one stack can be specified.

**row**

Is a subscript that specifies the first stack row to add to the data source.

; Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the semicolon, see *Terminating Command Syntax* on page 24.

**Example:** Adding Data From Multiple Stack Rows

The following example tries to add the data in rows 2, 3, and 4 of Stkemp into the data source. The stack subscript instructs the system to start in the second row of the stack. The FOR 3 instructs the system to INCLUDE the next three rows.

```plaintext
FOR 3 INCLUDE Emp_ID FROM Stkemp(2);
```
**Example:** Preventing Duplicate Records

You can execute the INCLUDE command after a MATCH command that fails to find a matching record. For example:

```sql
MATCH Emp_ID FROM Newemp;
ON NOMATCH INCLUDE Emp_ID FROM Newemp;
```

The INCLUDE command can also be issued without a preceding MATCH. In this situation the key field values are taken from the source stack or Current Area and a MATCH is performed internally. When a set of rows is input without a prior confirmation that it does not already exist in the data source, one or more of the rows in the stack may be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. For all of the rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work and a ROLLBACK issued if the entire set was not accepted. After an INCLUDE, the transaction variable FocError is given a value. If the INCLUDE is successful, FocError is zero. If the INCLUDE fails (for example, if the key values already exist in the data source), Maintain assigns a non-zero value to FocError, and (if the include was set-based) assigns the value of the row that failed to FocErrorRow. If at COMMIT time there is a concurrency conflict, Maintain sets FocCurrent to a non-zero value.

**Example:** Adding Multiple Segments

This example shows how data is added from two segments in the same path. The data comes from a stack named EmpInfo and the entire stack is used. When the INCLUDE is complete, the variable FocError is checked to see if the INCLUDE was successful. If it failed, a general error handling function is called:

```sql
FOR ALL INCLUDE Emp_ID Dat_Inc FROM EmpInfo;
IF FocError NE 0 THEN PERFORM Errhandle;
```

**Example:** Adding Data From the Current Area

The user is prompted for the employee ID and name. The data is included if it does not already exist in the data source. If the data already exists, it is not included, and the variable FocError is set to a non-zero value. Since the procedure does not check FocError, no error handling takes place and the user does not know whether or not the data is added:

```sql
NEXT Emp_ID Last_Name First_Name;
INCLUDE Emp_ID;
```

**Reference:** Usage Notes for INCLUDE

- If there is a FOR prefix, a stack must be mentioned in the FROM phrase.
When an INCLUDE command is complete, the variable FocError is set. If the INCLUDE is successful (the records to be added do not exist in the data source), then FocError is set to zero. If the records do exist, FocError is set to a non-zero value, and (if it is a set-based INCLUDE) FocErrorRow is set to the number of the row that failed.

Maintain requires that data sources to which it writes have unique keys.

**Reference:** Commands Related to INCLUDE

- **COMMIT** makes permanent all data source changes since the last COMMIT.
- **ROLLBACK** cancels all data source changes made since the last COMMIT.

**Data Source Position**

A Maintain procedure always has a position either within a segment or just prior to the first segment instance. If data has been retrieved, the position is the last record successfully retrieved on that segment. If a retrieval operation fails, the data source position remains unchanged.

If an INCLUDE is successful, the data source position is changed to the new record. On the other hand, if the INCLUDE fails, it might be because there is already a record in the data source with the same keys. In this case, the attempted retrieval prior to the INCLUDE is successful, and the position is on that record. Therefore, the position in the data source changes.

**Null Values**

If you add a segment instance that contains fields for which no data has been provided, and those fields have been defined in the Master File:

- **With** the MISSING attribute, they are assigned a null value.
- **Without** the MISSING attribute, they are assigned a default value of a space (for character and date and time fields) or zero (for numeric fields).

**INFER**

Stacks are array variables containing rows and columns. When defining a stack and its structure, provide a name for the stack and a name, format, and order for each of the columns in the stack.
Stacks can be defined in two ways:

- Performing actual data retrieval with the NEXT command, the stack is defined and populated at the same time. The stack is defined with all the segments that are retrieved. This is convenient when the procedure is processing on the same physical platform as the data source.

- If the procedure referring to a stack does not retrieve data, you must issue the INFER command to define the structure of the stack. When you issue the command, you specify a data source path. INFER defines the stack with columns corresponding to each field in the specified path. The Master File provides the names and formats of the columns. INFER may only be used to define stack columns that correspond to data source fields. To define user-defined variables, use the COMPUTE command.

A procedure that includes an INFER command must specify the name of the corresponding Master File in the MAINTAIN command, and must have access to the Master File.

**Syntax: How to Use the INFER Command**

The syntax of the INFER command is

```
INFER path_spec INTO stackname [;]
```

where:

- `path_spec`
  
  Identifies the path to be defined for the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical. Simply specify the segment once. (For paths with multiple segments, if you wish to make the code clearer to readers, you can also specify segments between the anchor and target.)

  To specify a segment, provide the name of the segment or of a field within the segment.

- `stackname`
  
  Is the name of the stack that you wish to define.

- `;`
  
  Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.
**Example:** Inferring Two Stacks

In the following called procedure, two INFER commands define the EmpClasses and ClassCredits stacks:

```
MAINTAIN FROM EmpClasses INTO ClassCredits
INFER Emp_ID Ed_Hrs Date_Attend Course_Code INTO EmpClasses;
INFER Emp_ID Course_Code Grade Credits INTO ClassCredits;
```

**Reference:** Commands Related to INFER

- **CALL** can be used to call one Maintain procedure from another.
- **COPY** can be used to copy data from one stack to another.
- **COMPUTE** can be used to define the contents of a stack for non-data source fields.

**Defining Non-Data Source Columns**

To define stack columns in a procedure for non-data source fields (fields created with the COMPUTE command), you do not need to provide a value for the column. The syntax is:

```
COMPUTE stackname.target_variable/format = ;
```

Note that the equal sign is optional when the COMPUTE is issued solely to establish format.

In the following example, the stack column TempEmp was passed to the called procedure. The COMPUTE is issued in the called procedure to define the variable prior to use:

```
COMPUTE EmpClasses.TempEmp_ID/A9 ;
```

**MAINTAIN**

The MAINTAIN command marks the beginning of a Maintain procedure. You can identify any data sources the procedure will access using the FILE phrase. If the request is to be called from another procedure, you can identify variables to be passed from and to the calling procedure using the FROM and INTO phrases.
**Syntax:** How to Use the MAINTAIN Command

The syntax of the MAINTAIN command is

```
MAINTAIN [FILE[S] filelist] [FROM varlist] [INTO varlist]
    filelist:filedesc [{AND|,} filedesc ...]
    varlist: {variable} [{variable} ...]
```

where:

MAINTAIN

Identifies the beginning of a Maintain request. It must be coded in uppercase letters.

FILE[S]

Indicates that the procedure accesses Master Files. The S can be added to FILE for clarity. The keywords FILE and FILES may be used interchangeably.

You access a Master File when you read or write to a data source, and when you use an INFER command to define the data source columns of a stack. For example, when you redefine a stack that has been passed from a parent procedure.

FROM

Is included if this procedure is called by another procedure, and that procedure passes one or more variables.

INTO

Is included if this procedure is called by another procedure, and this procedure passes one or more variables back to the calling procedure.

filelist

Is the names of the Master Files this procedure accesses.

filedesc

Is the name of the Master File that describes the data source that is accessed in the procedure.

AND

Is used to separate Master File names.

,

Is used to separate Master File names.

varlist

Is the variables, both scalar variables and stacks, which are passed to or from this procedure. Multiple variables are separated by blank spaces.
variable

Is the name of a scalar variable or stack. You can pass any variable except for those defined as variable-length character (that is, those defined as TX or A0) and those defined using STACK OF.

Reference: Usage Notes for MAINTAIN

- To access more than one data source, you can specify up to 15 Master Files per MAINTAIN command. If you must access more than 15 data sources, you can call other procedures that can each access an additional 15 data sources.

- There is a limit of 64 segments per procedure for all referenced data sources, although additional procedures can reference additional segments.

Reference: Commands Related to MAINTAIN

- END terminates the execution of a Maintain procedure.

- CALL is used to call one procedure from another.

Specifying Data Sources With the MAINTAIN Command

The MAINTAIN command does not require that any parameters are supplied. This means that Maintain procedures do not need to access data sources or stacks. You can use a procedure as a subroutine when sharing functions among different procedures, or when certain logic is not executed very frequently. For example, to begin a procedure that does not access any data sources and does not have any stacks as input or output, you simply begin the procedure with the keyword MAINTAIN.

However, the keyword FILE and the name of the Master File are required if you want to access a data source. The following example accesses the Employee data source:

MAINTAIN FILE Employee

A Maintain procedure can access several data sources by naming the corresponding Master Files in the MAINTAIN command:

MAINTAIN FILES Employee AND EducFile AND JobFile

Calling a Procedure From Another Procedure

You can use the CALL command to pass control to another procedure. When the CALL command is issued, control is passed to the named procedure. Once that procedure is complete, control returns to the item that follows the CALL command in the calling procedure.
Called procedures can also reside on remote WebFOCUS Servers, allowing you to partition
the logic of your application between machines.

For information about the CALL command, see CALL on page 64.

**Example: Passing Variables Between Procedures**

You can pass stacks and variables between procedures by using FROM and INTO variable
lists. In the following example, when the CALL Validate command is reached, control is passed
to the procedure named Validate along with the Emps stack. Once Validate is complete, the
data in the stack ValidEmps is sent back to the calling procedure. Notice that the calling and
called procedures both have the same FROM and INTO stack names. Although this is not
required, it is good practice to avoid giving the same stacks different names in different
procedures.

The calling procedure contains:

```plaintext
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Emps;
INFER emp_id into ValidEmps;
CALL Validate FROM Emps INTO ValidEmps;
.
.
.
END
```

The called procedure (Validate) contains:

```plaintext
MAINTAIN FILE Employee FROM Emps INTO ValidEmps
.
.
.
END
```

**MATCH**

The MATCH command enables you to identify and retrieve a single segment instance or path
instance by key value. You provide the key value using a stack or the Current Area. MATCH
finds the first instance in the segment chain that has that key.

You specify which path to retrieve by identifying its anchor and target segments. If the anchor
segment is not the root, you must establish a current instance in each of the ancestor segments
of the anchor. This enables MATCH to navigate from the root to the anchor segment instance.

The command always matches on the full key. If you wish to match on a partial key, use the
NEXT command and identify the value of the partial key in the WHERE phrase of the command.
If the data source has been defined without a key, you can retrieve a segment instance or path using the NEXT command, and identify the desired instance using the WHERE phrase of the command.

**Syntax:**

**How to Use the MATCH Command**

The syntax of the MATCH command is

```plaintext
MATCH path_spec [FROM stack[(row)]] [INTO stack[(row)]] [;]
```

where:

- **path_spec**
  - Identifies the path to be read from the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical. Specify the segment once. (For paths with multiple segments, if you wish to make the code clearer to readers, you can also specify segments between the anchor and target.)
  
  To specify a segment, provide the name of the segment or of a field within the segment.

- **FROM**
  - Is used to specify a stack containing a key value on which to match. If you omit this, Maintain uses a value in the Current Area. In either case, the columns containing the key value must have the same names as the corresponding key fields in the data source.

- **INTO**
  - Is used to specify the stack that the data source values are to be placed into. Values retrieved by MATCH are placed into the Current Area when an INTO stack is not supplied.

- **stack**
  - Is a stack name. Only one stack can be specified for each FROM or INTO phrase. The stack name should have a subscript specifying which row is to be used. If a stack is not specified, the values retrieved by the MATCH go into the Current Area.

- **row**
  - Is a subscript that specifies which row is used. The first row in the stack is matched against the data source if the FROM stack does not have a subscript. The data is placed in the first row in the stack if the INTO stack does not have a subscript.

- **;**
  - Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.
Example: Matching Keys in the Employee Data Source

The following example performs a MATCH on the key field in the PayInfo segment. It gets the value for Pay_Date from the Pay_Date field, which is in the Current Area. After the match is found, all of the field values in the PayInfo segment are copied from the data source into the Current Area:

MATCH Pay_Date;

The next example shows a MATCH on the key in the EmpInfo segment. It gets the value for Emp_ID from the Emp_ID column in the Cnt row of the Stackemp stack. After the match is found, all of the fields in the EmpInfo segment are copied into the Current Area:

MATCH Emp_ID FROM Stackemp(Cnt);

The last example is the same as the previous example except an output stack is mentioned. The only difference in execution is that after the match is found, all of the fields in the EmpInfo segment are copied into a specific row of a stack rather than into the Current Area:

MATCH Emp_ID FROM Stackemp(Cnt) INTO Empout(Cnt);

Reference: Commands Related to MATCH

- **NEXT** starts at the current position and moves forward through the data source. NEXT can retrieve data from one or more records.

- **REPOSITION** changes the data source position to be at the beginning of the chain.

How the MATCH Command Works

When a MATCH command is issued, Maintain tries to retrieve a corresponding record from the data source. If there is no corresponding value and an ON NOMATCH command follows, the command is executed.

The MATCH command looks through the entire segment to find a match. The NEXT command with a WHERE qualifier also locates a data source record, but does it as a forward search. That is to say, it starts at its current position and moves forward. It is not an exhaustive search unless positioned at the start of a segment. This can always be done with the REPOSITION command. A MATCH is equivalent to a REPOSITION on the segment followed by a NEXT command with a WHERE phrase specifying the key. If any type of test other than the equality test that the MATCH command provides is needed, the NEXT command should be used.
MNTCON CDN_FEXINPUT

By default, you must use a decimal point to indicate a decimal position when writing a value in a Maintain procedure (for example, a COMPUTE statement), and a comma if you wish to demarcate thousands, regardless of the CDN setting.

To write the value in a procedure using the format matching the CDN setting for a value other than OFF (for example, ON, QUOTE, QUOTEP, SPACE), use MNTCON CDN_FEXINPUT ON in the EDASPROF file or user profile, and use quotation marks to delimit the value. You can use single or double quotes when CDN=ON or SPACE. You must use double quotes when CDN=QUOTE or QUOTEP.

Example 1:
The following are both correct for all CDN settings by default:

```
COMPUTE MYVAL/D12.2=1234.56;
COMPUTE MYVAL/D12.2="1,234.56";
```

Example 2:
The following are both correct for SET CDN = QUOTE when the EDASPROF or user profile contains MNTCON CDN_FEXINPUT ON:

```
COMPUTE MYVAL/D12.2="1'234,56";
COMPUTE MYVAL/D12.2="1234,56";
```

Note: This command does not apply to values entered in a form at run time.

This command is outside the Maintain language, but is described in this chapter for your convenience.

Syntax: How to Use the MNTCON CDN_FEXINPUT Command

When using a CDN value other than OFF, place the following statement in the server profile file (edasprof.prf) or user profile:

```
MNTCON CDN_FEXINPUT {ON OFF}
```

where:

ON

- Allows you to write values in Maintain procedures in the manner used by the actual CDN setting, for example:
  - Using a comma to denote a decimal place when CDN=ON, SPACE, or QUOTE.
Using a single quote to demarcate thousands when CDN=QUOTE or QUOTEP.

- Using a space to demarcate thousands when CDN=SPACE.

Follow these rules when writing values using Continental Decimal Notation with MNTCON CDN_FEXINPUT ON:

- You must use double or single quotes to delimit values for ON or SPACE.
- You must use double quotes to delimit values for QUOTE. You must also use double quotes to delimit values for QUOTEP if you wish to write the value with single quotes to separate thousands.

**OFF**

Requires you to write values in Maintain procedures using a period to denote a decimal place for all CDN settings. OFF is the default value.

When demarcating thousands, a comma must be used, and the value must be enclosed in quotation marks.

**MNTCON COMPILE**

The MNTCON COMPILE command creates a compiled Maintain procedure which, under Windows and UNIX, has an extension of .fcm., and under z/OS is allocated to ddname FOCCOMP.

You can reduce the time needed to start a Maintain procedure that contains forms by compiling the procedure. The more frequently the Maintain procedure will be run, the more time you save by compiling it.

This command is outside the Maintain language, but is described here in Command Reference for your convenience.

**Syntax:**

**How to Use the MNTCON COMPILE Command**

The syntax of the MNTCON COMPILE command is

```
MNTCON COMPILE [dirname/]proclname
```

where:

`dirname`

Is the directory name on the WebFOCUS Server where the Maintain procedure is located. This is optional.
**procname**

Is the name of a Maintain procedure. First, the MNTCON COMPILE command looks for a Maintain procedure with a .mnt extension or a MAINTAIN file type or ddname. If it doesn’t find one, it looks for a Maintain procedure with a .fex extension or a FOCEXEC file type or ddname.

**Reference:** Commands Related to MNTCON COMPILE

- **MNTCON RUN** executes compiled Maintain procedures.
- **MNTCON EX** executes uncompiled Maintain procedures.

**MNTCON EX**

You use the MNTCON EX command to run an uncompiled Maintain procedure.

This command is outside the Maintain language, but is described here in Command Reference for your convenience.

**Syntax:** How to Use the MNTCON EX Command

To run an uncompiled WebFOCUS Maintain procedure (with either a .mnt or .fex extension, or a MAINTAIN or FOCEXEC file type or ddname), use the following syntax:

```bash
MNTCON EX [dirname/]procname [-v parm1, ... parmn]
```

where:

- **dirname**
  
  Is the directory name on the WebFOCUS Server where the Maintain procedure is located. This is optional.

- **procname**
  
  Is the name of a Maintain procedure. First, the MNTCON EX command looks for a Maintain procedure with a .mnt extension or a MAINTAIN file type or ddname. If it doesn’t find one, it looks for a Maintain procedure with a .fex extension or a FOCEXEC file type or ddname.

- **-v**
  
  Is the flag that indicates parameters will be passed to the Maintain procedure. This is optional.
Can be either positional parameters or parm="value" keyword parameters. Parameter types can be mixed within the same MNTCON EX command line. The maximum number of parameters you can pass is 128. You should separate all parameters using commas. You should use single or double quotes to enclose values containing spaces or commas. Use with Sys_mgr functions (Sys_Mgr.Get_NameParm, Sys.Mgr.Get_InputParams_Count and Sys_Mgr.Get_PositionParm) to retrieve the values. If any of these functions are unsuccessful, FOCERROR is set to -1.

For more information, see the Developing WebFOCUS Maintain Applications manual, and SYS_MGR on page 181.

Reference: Commands Related to MNTCON EX

- MNTCON COMPILE compiles Maintain procedures.
- MNTCON RUN executes compiled Maintain procedures.

Invoking Maintain Procedures: Passing Parameters

You can issue MNTCON EX or MNTCON RUN with the flag –v to pass input parameters from the command line when invoking Maintain applications, in a manner similar to passing parameters to WebFOCUS FOCEXECs. This method bypasses the requirement of importing the webbase2 file and coding web client variable retrieval. You can use this syntax in a FOCEXC (fex) or from within a backend server edastart –t session.

Positional and key-matching parameters are supported, and you can use both together in the same Maintain EX or RUN command. Parameters are defined as A0. The maximum number of parameters you can pass is 128. You may include Dialog Manager commands in a FOCEXC when invoking MNTCON EX or RUN with the –v option.

Syntax: How to Use the MNTCON EX Command to Pass Parameters

MNTCON [EX|RUN] proname –v "parm1value" ... "parm_nvalue"

where:

parm1value ... parm_nvalue

Can be either positional parameter values in single quotes or double quotes, or a parm="value" key-matching parameter. You can mix positional and key-matching parameters.
The target Maintain procedure uses Maintain SYS_MGR function subcommands to retrieve the values.

Sys_mgr.get_positionParm
Sys_mgr.GET_inputparams_count
Sys_mgr.get_nameParm

If any of these SYS_MGR functions is not successful, FOCERROR is set to -1.

For more information on SYS_MGR functions, see SYS_MGR on page 181.

**Example: Passing and Retrieving Parameters**

MNTCON EX START1 -v abc, '24 Houston Center', ADDR='Cape Canaveral', COUNTRY=USA

Target Maintain procedure START1 could include:

Parm1/a0=sys_mgr.get_positionParm(1);

to get the first positional parameter. Here it returns value abc for Parm1.

Posvar/i2=sys_mgr.GET_inputparams_count();

to return the total number of positional parameters. Here it returns 2 for Posvar.

Address/a0=sys_mgr.get_nameParm('ADDR');

to return value for key-matching parameter ADDR. Here it returns Cape Canaveral for Address.

**Note:** Sys_mgr.get_nameParm is case-sensitive. Use the same case for the parameter when retrieving the value as you use when passing it.

For more information on SYS_MGR functions, see SYS_MGR on page 181.

**MNTCON EXIT WARNING**

By default, the exit message, *This application has been disconnected*, appears when a browser session containing an active Maintain application is closed. To control the display of this exit warning, use MNTCON EXIT_WARNING.

This command is outside the Maintain language, but is described in this chapter for your convenience.
How to Use the MNTCON EXIT_WARNING Command

Syntax:

Place the following statement in the server profile file (edasprof.prf) or user profile:

```sql
MNTCON EXIT_WARNING {ON|OFF}
```

where:

**ON**

Enables the display of the exit message, *This application has been disconnected*, when an active Maintain browser is closed. ON is the default value.

**OFF**

Disables the exit warning that displays when an active Maintain browser is closed.

MNTCON LOADIMAGE

The MNTCON LOADIMAGE command loads all of the Maintain procedures in a WebFOCUS Maintain application prior to running the application on a Shared Application Server. These Maintain procedures must be compiled.

This command is outside the Maintain language, but is described in this chapter for your convenience.

Syntax: How to Use the MNTCON LOADIMAGE Command

To load all of the (compiled) procedures for a WebFOCUS Maintain application for use on a Shared Application Server, place the following statement in the server profile file (edasprof.prf) or user profile:

```sql
MNTCON LOADIMAGE [dirname1/]procl.
```

where:

**dirname1/dirmamen**

Are the names of the directories on the server where you can find the application’s Maintain procedures.

**procl / procn**

Are the names of all of the application’s Maintain procedures. The names must be uppercase.
These commands must appear in one of the following:

- The global profile or service profile of the server instance.
- A WebFOCUS procedure, residing in the EDASYNR path of the server instance, which is then invoked from the profile with -INCLUDE.

**Reference:** Commands Related to MNTCON LOADIMAGE

- **MNTCON COMPIL**E compiles Maintain procedures.
- **MNTCON RUNIMAGE** runs a compiled procedure for the Shared Application Server.

**MNTCON MATCH_CASE**

By default, segment names and field names in Master Files must be in uppercase. To enable mixed-case names, use MNTCON MATCH_CASE ON.

**Syntax:** How to Enable Mixed-Case Naming

The feature to support mixed-case and NLS characters in the Master File is enabled by the following command in the EDASPROF, user, group, or service profile:

```plaintext
MNTCON MATCH_CASE ON|OFF
```

where:

- **OFF**
  - Is the default. Segment names and field names in Master Files still must be uppercase, and Maintain refers to them in mixed-case or lowercase without error.

- **ON**
  - Means that mixed-case and NLS characters will be respected. Developers must be consistent in their references to named components in terms of the case used.

  Components are:

  - Case names
  - Class names
  - Function names
  - Object names
  - Stack names
  - Variable name
With the feature enabled (MNTCON MATCH_CASE ON in a profile):

- Field names COUNTRY, Country and CounTry all refer to different fields.
- Developers must be consistent with casing when referring to classes, functions, objects, stacks, and variables. For example, Case MYCASE would need any associated Perform statement to refer to the casename as MYCASE.
- IWC. function names must be written exactly as follows:
  - IWC.putCgiData
  - IWC.getCgiData
- MAINTAIN and END are still required to be in uppercase when developing using the MDE.
- Certain keywords will be automatically translated to uppercase. For example, contains and CONTAINS will always mean the same thing.

**Note:** With the feature off, there should be no issues running previously-developed applications.

With the feature enabled, previously-written applications would need to be reviewed, then updated (to keep the case, class, and other names consistent), and finally redeployed.

**MNTCON PERFORMANCE_ANALYSIS**

To assist in the process of performance tuning for a WebFOCUS Maintain application, you can track the start and end of Maintain subprocedures invoked using CALL and EXEC without having to collect individual traces. Use MNTCON PERFORMANCE_ANALYSIS ON to generate just two lines of output to the EDAPRINT.LOG for each CALL and EXEC, showing the start and end times. This does not heavily tax server performance, even when tracking many users. This can help you pinpoint which procedures are most heavily called within an application. You can easily extract the format to FOCUS reports for analysis.

This command is outside the Maintain language, but is described in this chapter for your convenience.

**Syntax:**  
How to Enable Maintain Performance Analysis on the Server

Use the following statement in the server profile file (edasprof.prf) or in individual user profiles:

```
MNTCON PERFORMANCE_ANALYSIS [ON|OFF]
```
where:

**ON**

Generates a timestamp in the EDAPRINT.LOG file for the start and completion of Maintain CALL and EXEC procedures.

**OFF**

Disables the performance analysis feature. OFF is the default value.

### MNTCON RADIO_BUTTON_EMIT_TEXT

When using a web link event for a radio button control, the selected item is passed as a text value. To send FOCINDEX instead of a text value, use MNTCON RADIO_BUTTON_EMIT_TEXT.

This command is outside the Maintain language, but is described in this chapter for your convenience.

**Syntax:**  
How to Send FOCINDEX From a Radio Button Web Link Event

Use the following statement in the server profile file (edasprof.prf) or in individual user profiles:

```
MNTCON RADIO_BUTTON_EMIT_TEXT {ON|OFF}
```

where:

**ON**

Indicates that the text value of the selected item of a radio button will be passed in a web link event. ON is the default value.

**OFF**

Indicates that FOCINDEX will be used.

### MNTCON REMOTESTYLE

By default, Maintain supports the use of a variable for the server name in CALL AT and EXEC AT statements. It is possible to disable the variable server name feature by using MNTCON REMOTESTYLE.

This command is outside the Maintain language, but is described in this chapter for your convenience.

**Syntax:**  
How to Disable the Variable Server Name Feature

Use the following statement in the server profile file (edasprof.prf) or in individual user profiles:

```
MNTCON REMOTESTYLE {ON|OFF}
```
where:

**ON**

Allows variables to be used in the AT SERVER syntax for use with the CALL and EXEC commands. ON is the default value.

**OFF**

Disables the variable server name feature.

### MNTCON RUN

You use the MNTCON RUN command to run a Maintain procedure that has been compiled. This command is outside the Maintain language, but is described here for your convenience.

**Syntax:** How to Use the MNTCON RUN Command

To run a Maintain procedure that has been compiled (with either a .fcm extension or a FOCCOMP file type or ddname), use the following syntax

```plaintext
MNTCON RUN [dirname/]procname [-v parm1 , ... parmn]
```

where:

`dirname`

Is the directory name on the WebFOCUS Server where the Maintain procedure is located. This is optional.

`procname`

Is the name of a Maintain procedure.

`-v`

Is the flag that indicates parameters will be passed to the Maintain procedure. This is optional.

`parm1 ... parmn`

Can be either positional parameters or parm="value" keyword parameters. Parameter types can be mixed within the same MNTCON RUN command line. The maximum number of parameters you can pass is 128. You should separate all parameters using commas. You should use single or double quotes to enclose values containing spaces or commas. Use with Sys_mgr functions (Sys_Mgr.Get_NameParm, Sys.Mgr.Get_InputParams_Count and Sys_Mgr.Get_PositionParm) to retrieve the values. If any of these functions are unsuccessful, FOCERROR is set to -1.
For more information, see the Developing WebFOCUS Maintain Applications manual, and SYS_MGR on page 181.

Reference: Commands Related to MNTCON RUN

- **MNTCON COMPILE** executes compiled Maintain procedures.
- **MNTCON EX** executes uncompiled Maintain procedures.

For information on passing parameters with MNTCON RUN, see *Invoking Maintain Procedures: Passing Parameters* on page 132.

Syntax: How to Use the MNTCON RUN Command to Pass Parameters

```
MNTCON [EX|RUN] procname -v "parm1value" ... "parmNvalue"
```

where:

```
parm1value ... parmNvalue
```

Can be either positional parameter values in single quotes or double quotes, or a `parm="value"` key-matching parameter. You can mix positional and key-matching parameters.

The target Maintain procedure uses Maintain SYS_MGR function subcommands to retrieve the values.

```
Sys_mgr.get_positionParm
Sys_mgr.GET_inputparams_count
Sys_mgr.get_nameParm
```

If any of these SYS_MGR functions is not successful, FOCERROR is set to -1.

For more information on SYS_MGR functions, see SYS_MGR on page 181.

Example: Passing and Retrieving Parameters

```
MNTCON RUN START1 -v NASA, '24 Houston Center', ADDR='Cape Canaveral', COUNTRY=USA
```

Maintain procedure START1 could include:

```
Parm1/a0=sys_mgr.get_positionParm(1);
```

to get the first positional parameter. Here it returns value NASA for Parm1.

```
Parm1/a0=sys_mgr.get_positionParm(3);
```
Here FOCERROR is set to -1, as there were only two positional parameters passed.

\[\text{Posvar/i2} = \text{sys_mgr.GET_inputparams_count();}\]

to return the total number of positional parameters. Here it returns 2 for Posvar.

\[\text{Address/a0} = \text{sys_mgr.get_nameParm('ADDR');}\]

to return value for key-matching parameter ADDR. Here it returns Cape Canaveral for Address.

**Note:** Sys_mgr.get_nameParm is case-sensitive. Use the same case for the parameter when retrieving the value as you use when passing it.

For more information on SYS_MGR functions, see SYS_MGR on page 181.

**MNTCON RUNIMAGE**

The MNTCON RUNIMAGE command runs a compiled Maintain procedure on a Shared Application Server. These Maintain procedures must be compiled, and they must have been loaded in the server instance profile.

This command is outside the Maintain language, but is described in this chapter for your convenience. For more information on the Shared Application Server, see the Developing WebFOCUS Maintain Applications manual.

**Syntax:** How to Use the MNTCON RUNIMAGE Command

To run a compiled WebFOCUS Maintain procedure on a Shared Application Server, use the following syntax

\[\text{MNTCON RUNIMAGE [dirname/]procname}\]

where:

- **dirname**
  - Is the directory name on the WebFOCUS Server where the Maintain procedure is located.
  - This is optional.

- **procname**
  - Is the name of a Maintain procedure.

**Reference:** Commands Related to MNTCON RUNIMAGE

- **MNTCON COMPIL**e executes compiled Maintain procedures.

- **MNTCON LOADIMAGE** loads a compiled procedure for the Shared Application Server.
MODULE

The MODULE command accesses a source code library so the current procedure can use the class definitions of the library and Maintain functions. (A library is a nonexecutable procedure, and is implemented as a project component called an import module.)

Syntax: How to Use the MODULE Command

The MODULE command must immediately follow the MAINTAIN command. The syntax of the MODULE command is

```
MODULE IMPORT (library_name [, library_name] ... );
```

where:

- `library_name`
  Is the name of the library that you wish to import as a source code library. Specify its file name without an extension. The file must reside in the path defined by the EDASYNR environment variable.

  If a library is specified multiple times in a MODULE command, Maintain will include the library only once in order to avoid a loop.

Reference: Commands Related to MODULE

- **DESCRIBE** defines classes. You can use DESCRIBE to include classes in a library.
- **CASE** defines a function. You can use CASE to include functions in a library.

What You Can and Cannot Include in a Library

You can include most Maintain language commands and structures in a library. However, there are some special opportunities and restrictions of which you should take note:

- **Other libraries.** You can place one library within another, and can nest libraries to any depth. For example, to nest library B within library A, issue a MODULE IMPORT B command within library A.

  If a given library is specified more than once in a series of nested libraries, Maintain will only include the library once in order to avoid a loop.

- **Top function.** Because a library is a nonexecutable procedure, it has no Top function.
- **Forms.** A library cannot contain forms.
- **Data sources.** A library cannot refer to data sources. For example, it cannot contain data source commands (such as NEXT and INCLUDE) and cannot refer to data source stacks.

**NEXT**

The NEXT command selects and reads segment instances from a data source. You can use NEXT to read an entire set of records at a time, or just a single segment instance. You can select segments by field value or sequentially.

You specify a path running from an anchor segment to a target segment. NEXT reads all the fields from the anchor through the target, and (if the anchor segment is not the root) all the keys of the ancestor segments of the anchor. It copies what it has read to the stack that you specify or, if you omit a stack name, to the Current Area.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor. This enables NEXT to navigate from the root to the current instance of the anchor segment.

In each segment that it reads, NEXT works its way forward through the segment chain. When no more records are available, the NONEXT condition arises and no more records are retrieved unless the procedure issues a REPOSITION command. REPOSITION causes a reposition to just prior to the beginning of the segment chain. If you are familiar with the SQL language, the NEXT command acts as a combination of the SQL commands SELECT and FETCH, and allows you to use the structure of the data source to your advantage when retrieving data.

**Syntax:** **How to Use the NEXT Command**

The syntax of the NEXT command is

```
[FOR {int|ALL}] NEXT path [INTO stack[(row)]] [WHERE where_expression1 [AND where_expression2 ...]] [;]
```

where:

**FOR**

Is a prefix that is used with int or ALL to specify how many data source records are to be retrieved. If FOR is not specified, NEXT works like FOR 1 and the next record is retrieved. If the FOR phrase is used, the INTO phrase must also be used.

**int**

Is an integer constant or variable that specifies the number of data source records that are retrieved from the data source. Retrieval starts at the current position in the data source.
ALL
Specifies that starting at the current data source position, all data source segments referred to in the field list are examined.

**path**
Identifies the path to be read from the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical, simply specify the segment once. (For paths with multiple segments, if you wish to make the code clearer to readers, you can also specify segments between the anchor and target.)

To specify a segment, provide the name of the segment or of a field within the segment.

**INTO**
Is used with a stack name to specify the name of the stack into which the data source records are copied.

**stack**
Is the name of the stack that the data source values are placed into. Only one stack can be specified.

**row**
Is a subscript that specifies in which row of the stack the data source values are placed. If no subscript is provided, the data is placed in the stack starting with the first row.

**where_expression1, where_expression2**
Is any valid NEXT WHERE expression. You can use any valid relational expression, described in *Relational Expressions* on page 50. NEXT can also use some enhanced screening conditions not available in other situations. For more information, see *Using Selection Logic to Retrieve Rows* on page 146.

**;**
Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.
Reference: Usage Notes for NEXT

- If an INTO stack is specified, and that stack already exists, new rows are added starting at the row specified. If no stack row number is specified then data is added starting at the first row. In either case, it is possible that some existing rows may be written over. If a NEXT command causes only some of the rows in a stack to be overwritten, the rest of the stack remains intact. If the subscript provided on the INTO stack is past the end of the existing stack, the intervening rows are initialized to spaces, zeroes, or nulls (missing values) as appropriate. If the new stack overwrites some of the rows of the existing stack, only those rows are affected. The rest of the stack remains intact.

- If no FOR prefix is used and no stack name is supplied, the values retrieved by the NEXT command go into the Current Area.

Reference: Commands Related to NEXT

- REPOSITION changes the data source position to be at the beginning of the chain.
- MATCH searches the entire segment for a matching field value. It retrieves an exact match in the data source.

Copying Data Between Data Sources

You can use the NEXT command to copy data between data sources. It is helpful to copy data between data sources when transaction data is gathered by one application and must be stored for use by another application. It is also helpful when the transaction data is to be applied to the data source at a later time or in a batch environment.

Example: Copying Data to the Movies Data Source

For example, assume that you want to copy data from a fixed-format data source named FilmData into a FOCUS data source named Movies. You describe FilmData using the following Master File:

```
FILENAME=FILMDATA, SUFFIX=FIX
SEGNAMES=MOVINFO, SEGTYPE=S0
  FIELDNAME=MOVIECODE, ALIAS=MCOD, USAGE=A6, ACTUAL=A6,$
  FIELDNAME=TITLE, ALIAS=MTL, USAGE=A39, ACTUAL=A39,$
  FIELDNAME=CATEGORY, ALIAS=CLASS, USAGE=A8, ACTUAL=A8,$
  FIELDNAME=DIRECTOR, ALIAS=DIR, USAGE=A17, ACTUAL=A17,$
  FIELDNAME=RATING, ALIAS=RTG, USAGE=A4, ACTUAL=A4,$
  FIELDNAME=RELDATE, ALIAS=RDATE, USAGE=YMD, ACTUAL=A6,$
  FIELDNAME=WHOLESALEPR, ALIAS=WPRC, USAGE=F6.2, ACTUAL=A6,$
  FIELDNAME=LISTPR, ALIAS=LPRC, USAGE=F6.2, ACTUAL=A6,$
  FIELDNAME=COPIES, ALIAS=NOC, USAGE=I3, ACTUAL=A3,$
```
The fields in FilmData have been named identically to those in Movies to establish the correspondence between them in the INCLUDE command that writes the data to Movies.

You can read FilmData into Movies using the following procedure:

```
MAINTAIN FILE Movies AND FilmData
FOR ALL NEXT FilmData.MovieCode INTO FilmStack;
FOR ALL INCLUDE Movies.MovieCode FROM FilmStack;
END
```

All field names in the procedure are qualified to distinguish between identically-named fields in the input data source (FilmData) and the output data source (Movies).

**Loading Multi-Path Transaction Data**

When you wish to load data from a transaction data source into multiple paths of a data source, you should process each path independently. Use one pair of NEXT and INCLUDE commands per path.

For example, assume that you have a transaction data source named TranFile whose structure is identical to that of the VideoTrk data source.

If you wish to load the transaction data from both paths of TranFile into both paths of VideoTrk, you could use the following procedure:

```
MAINTAIN FILES TranFile AND VideoTrk
FOR ALL NEXT TranFile.CustID TranFile.ProdCode INTO ProdStack;
REPOSITION CustID;
FOR ALL NEXT TranFile.CustID TranFile.MovieCode INTO MovieStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.ProdCode FROM ProdStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.MovieCode FROM MovieStack;
END
```

Alternatively, if you choose to store each path of transaction data in a separate single-segment transaction data source, the same principles apply. For example, if the two paths of TranFile are stored separately in transaction data sources TranProd and TranMove, the previous procedure would change as highlighted below:

```
MAINTAIN FILES TranProd AND TranMove AND VideoTrk
FOR ALL NEXT TranProd.CustID INTO ProdStack;
FOR ALL NEXT TranMove.CustID INTO MovieStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.ProdCode FROM ProdStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.MovieCode FROM MovieStack;
END
```
Retrieving Multiple Rows: The FOR Phrase

The FOR phrase is used to specify the number of data source records that are to be retrieved. As an example, if FOR 10 is used, ten records are retrieved. A subsequent FOR 10 retrieves the next ten records starting from the last position. If an attempt to retrieve ten records only returns seven because the end of the chain is reached, the command retrieves seven records, and the ON NONEXT condition is raised.

The following retrieves the next ten instances of the EmpInfo segment and places them into Stackemp:

```plaintext
FOR 10 NEXT Emp_ID INTO Stackemp;
```

Using Selection Logic to Retrieve Rows

When you are retrieving rows using the NEXT command, you have the option to restrict the rows you retrieve using the WHERE clause. The syntax for this option is

```
WHERE operand1 comparison_op1 operand2
[AND operand3 comparison_op1 operand4 ...]
```

where:

- `operand1`, `operand2`, `operand3`, `operand4`, ...

  Are operands. In each NEXT WHERE expression, one operand must be a data source field, and one must be a valid Maintain expression that does not refer to a data source field.

  For more information about Maintain expressions, see Expressions Reference on page 27.

- `comparison_op1`, `comparison_op2`, ...

  Can be any of the comparison operators listed in Logical Operators on page 51 or any of the comparison operators listed in Comparison Operators on page 147. Some comparison operators may be listed in both places. This means that they can be used in a WHERE clause in an enhanced way.

The following example retrieves every instance of the EmpInfo segment that has a department value of MIS:

```plaintext
FOR ALL NEXT Emp_ID INTO EmpStack WHERE Department EQ 'MIS';
```

Literals can be enclosed in either single (') or double (") quotation marks. For example, the following produces exactly the same results as the last example:

```plaintext
FOR ALL NEXT Emp_ID INTO EmpStack WHERE Department EQ "MIS";
```
The ability to use either single or double quotation marks provides the added flexibility of being able to use either single or double quotation marks in text. For example:

NEXT Emp_ID WHERE Last_Name EQ "O'HARA";
NEXT Product WHERE Descr CONTAINS 'TEST';

This example starts at the beginning of the segment chain and searches for all employees that are in the MIS department. All retrieved segment instances are copied into a stack:

REPOSITION Emp_ID;
FOR ALL NEXT Emp_ID INTO Misdept WHERE Department EQ 'MIS';

After FOR ALL NEXT is processed, you are positioned at the end of the segment chain. Therefore, before issuing an additional NEXT command on the same segment chain, you should issue a REPOSITION command to be positioned prior to the first instance in the segment chain.

**Reference:** Comparison Operators

**IS, EQ, NE, IS_NOT**

Select data source values using wildcard characters (you embed the wildcards in a character constant in the non-data source operand). You can use dollar sign wildcards ($) throughout the constant to signify that any character is acceptable in the corresponding position of the data source value.

If you wish to allow any value of any length at the end of the data source value, you can combine a dollar sign wildcard with an asterisk ($*) at the end of the constant.

For example:

WHERE ZipCode IS '112$$'

**CONTAINS, OMITS**

Select data source values that contain or omit a character string stored in a variable.

For example, the following returns all data where the word BANK is part of the bank name:

COMPUTE name/A4 = 'BANK';
FOR ALL NEXT bank_code
bank_name into stackname
   WHERE bank_name CONTAINS name;

The following returns all data where the bank name does not include the word BANK:

COMPUTE name/A4 = 'BANK';
FOR ALL NEXT bank_code
bank_name into stackname

WHERE bank_name OMITS name;

EXCEEDS

Selects data source values that are greater than a numeric value.

For example:

WHERE TOTAL Curr_sal EXCEEDS 110000

IN (list), NOT_IN (list)

Select data source values that are in or not in a list. IN and NOT_IN can be used with all data types.

For example, the following returns all data where the bank name is not in the list:

FOR ALL NEXT emp_id bank_name INTO stackname
WHERE bank_name NOT_IN
('ASSOCIATED BANK', CITIBANK)

EQ_Mask, NE_Mask

Select data source values that match or do not match a mask.

Use the $ sign to replace each letter in the value. Masks can only be used with alphanumeric data. The masked value may be hard coded or a variable.

For example, the following returns all data where the bank code starts with AAA and has any character at the end:

COMPUTE code/A4='AAA$';
FOR ALL NEXT bank_name
INTO stackname
WHERE bank_code EQ_Mask code;

The following returns all data where the bank code does not match the mask:

COMPUTE code/A4='AAA$';
FOR ALL NEXT bank_name
INTO stackname
WHERE bank_code NE_Mask code;
Using NEXT After a MATCH

NEXT can also be used in conjunction with the MATCH command. This example issues a MATCH for employee ID. If there is not a match, a message is displayed. If there is a match, all the instances of the child segment for that employee are retrieved and placed in the stack Stackemp. The NEXT command can be coded as part of an ON MATCH condition, but it is not required, as the NEXT will only retrieve data based on the current position of higher level segments.

```
MATCH Emp_ID
ON NOMATCH BEGIN
   TYPE "The employee ID is not in the data source."
   GOTO Getmore;
ENDBEGIN
FOR ALL NEXT Dat_Inc INTO Stackemp;
```

Using NEXT for Data Source Navigation: Overview

The segments that NEXT operates on are determined by the fields mentioned in the NEXT command. The list of fields is used to determine the anchor segment (the segment closest to the root) and the target segment (the segment furthest from the root). Every segment starting with the anchor and ending with the target make up the scope of the NEXT command, including segments not mentioned in the NEXT command. Both the target and the anchor must be in one data source path.

NEXT does not retrieve outside the scope of the anchor and target segment. All segments not referenced remain constant, which is why NEXT can act like a “next within parent.”
As an example, look at a partial view of the Employee data source:

If a NEXT command has SalInfo as the anchor segment and the target is the Deduct segment, it also needs to retrieve data for the EmpInfo segment, which is the parent of the SalInfo segment based on its current position. The position for the EmpInfo segment can be established by either a prior MATCH or NEXT command. If no position has been established for the EmpInfo segment, an error occurs.

You can use the NEXT command for:

- Data Source Navigation: Using NEXT With One Segment on page 151.
- Data Source Navigation: Using NEXT With Multiple Segments on page 152.
- Data Source Navigation: Using NEXT Following NEXT or MATCH on page 154.
Data Source Navigation: Using NEXT With One Segment

If a NEXT references only one segment and has no WHERE phrase or FOR prefix, it always moves forward one instance within that segment. If the segment is not the root, all parent segments must have a position in the data source and only those instances descending from those parents are examined and potentially retrieved. The NEXT command starts at the current position within the segment, and each time the command is encountered, it moves forward one instance. If a prior position has not been established within the segment (no prior NEXT, MATCH, or REPOSITION command has been issued), the NEXT retrieves the first instance.

The following command references the root segment, so there is no parent segment in which to have a position:

```
NEXT Emp_ID;
```

The following command refers to a child segment, so the parents to this segment must already have a position and that position does not change during the NEXT operation:

```
NEXT Pay_Date;
```

If the NEXT command uses the FOR prefix, it works the same as described above, but rather than moving forward only one data source instance, it moves forward as many rows as the FOR specifies. The following retrieves the next three instances of the EmpInfo segment:

```
FOR 3 NEXT Emp_ID INTO Stemp;
```

If a FOR prefix is used, an INTO stack must be specified. However, an INTO stack can be specified without the FOR prefix.

If a WHERE phrase is specified and there is no FOR prefix, the NEXT moves forward as many times as necessary to retrieve one row that satisfies the selection criteria specified in the WHERE phrase. The following retrieves the next employee in the MIS department:

```
NEXT Emp_ID WHERE Department EQ 'MIS';
```

If the NEXT command does not have an INTO stack name, the output of the NEXT (the value of all of the fields in the segment instance) goes into the Current Area. If an INTO stack is specified, the output goes into the stack named in the command. If more than one row is retrieved by using a FOR prefix, the number of rows specified in the FOR are placed in the stack. If the INTO stack specifies a row number (for example, INTO Mystack(10)), then the rows are added to the stack starting with that row number. If the INTO stack does not specify a row number, the rows are added to the stack starting at the first row.

The following retrieves all of the fields from the next instance in the segment that Emp_ID is in and places the output into the first row of the Stemp stack:

```
NEXT Emp_ID INTO Stemp;
```
If the NEXT command has both a WHERE phrase and a FOR prefix, it moves forward as many times as necessary to retrieve the number of rows specified in the FOR phrase that satisfies the selection criteria specified in the WHERE phrase. The following retrieves the next three employees in the MIS department and places the output into the stack called Stemp:

```plaintext
FOR 3 NEXT Emp_ID INTO Stemp WHERE Department EQ 'MIS';
```

If there were not as many rows retrieved as you specified in the FOR prefix, you can determine how many rows were actually retrieved by checking the FocCount variable of the target stack.

**Data Source Navigation: Using NEXT With Multiple Segments**

If a NEXT command references more than one segment, each time the command is executed it moves forward within the target (the lowest level child segment). Once the target no longer has any more instances, the next NEXT moves forward on the parent of the target and repositions itself at the beginning of the chain of the child. In the following example, the REPOSITION command changes the position of EmpInfo to the beginning of the data source (EmpInfo is in the root). The first NEXT command finds the first instance of both segments. When the second NEXT is executed, what happens depends on whether there is another instance of the SallInfo segment, because the NEXT command does not retrieve short path instances (that is, it does not retrieve path instances that are missing descendant segments). If there is another instance, the second NEXT moves forward one instance in the SallInfo segment. If there is only one instance in the SallInfo for the employee retrieved in the first NEXT, the second NEXT moves forward one instance in the EmpInfo segment. When this happens, the SallInfo segment is positioned at the beginning of the chain and the first SallInfo instance is retrieved. If there is no instance of SallInfo, the NEXT command retrieves the next record that has a SallInfo segment instance.

```plaintext
REPOSITION Emp_ID;
NEXT Emp_ID Pay_Date;
NEXT Emp_ID Pay_Date;
```

When there is a possibility of short paths, and the intention is to retrieve the data from the parent even if there is no data for the child, NEXT should be used on one segment at a time, as described in *Data Source Navigation: Using NEXT Following NEXT or MATCH* on page 154. If a NEXT command uses the FOR n prefix, it works the same as described above, but rather than moving forward only one data source instance, it moves forward as many records as are required to retrieve the number specified in the FOR prefix.

For instance, the following command retrieves the next five instances of the EmpInfo and SallInfo segments and places the output into the Stemp stack. The five records may or may not all have the same EmpInfo segment:

```plaintext
FOR 5 NEXT Emp_ID Dat_Inc INTO Stemp;
```
If the data source is populated as follows,

1. EMP_ID 071382660, DAT_INC 820101
2. EMP_ID 071382660, DAT_INC 810101
3. EMP_ID 112847612, DAT_INC 820101
4. EMP_ID 117593129, DAT_INC 820601
5. EMP_ID 117593129, DAT_INC 820501

all of the fields from the following segment instances are added to the stack:

If a WHERE phrase is specified, the NEXT moves forward as many times as necessary to retrieve one record that satisfies the selection criteria specified in the WHERE phrase. For example, the following retrieves the next record where the child segment has the field Gross greater than 1,000:

```
NEXT Emp_ID Pay_Date WHERE Gross GT 1000;
```

If both a WHERE phrase and a FOR prefix are specified, the NEXT moves forward as many times as necessary to retrieve the number specified in the FOR prefix that satisfies the selection criteria specified in the WHERE phrase. For instance, the following retrieves all of the records where the Gross field is greater than 1,000. As stated above, if more than one segment is mentioned and there is a FOR prefix, the data retrieved may come from more than one employee:

```
FOR ALL NEXT Emp_ID Pay_Date INTO Stemp WHERE Gross GT 1000;
```
If the NEXT command does not have an INTO stack name provided, the output of the NEXT is copied into the Current Area. If an INTO stack is specified, the output is copied into the stack named in the command. The number of records retrieved is the number that is placed in the stack. If the INTO stack specifies a row number (for example, INTO Mystack(10)) then the records are added to the stack starting at the row number. If the INTO stack does not specify a row number, the rows are added to the stack starting with the first row in the stack. If data already exists in any of the rows, those rows are cleared and replaced with the new values.

If the NEXT command can potentially retrieve more than one record (the FOR prefix is used), an INTO stack must be specified. If no stack is provided, an error message is displayed and the procedure is terminated.

**Data Source Navigation: Using NEXT Following NEXT or MATCH**

In order to use NEXT through several segments, specify all the segments in one NEXT command or use several NEXT commands. If all of the segments are placed into one NEXT command, there is no way to know when data is retrieved from a parent segment and when it is retrieved from a child. To have control over when each segment is retrieved, each segment should have a NEXT command of its own. In this way, the first NEXT establishes the position for the second NEXT.

A NEXT command following a MATCH command works in a similar way. The first command (MATCH) establishes the data source position.

In the following example, the REPOSITION command places the position in the EmpInfo segment and all of its children to the beginning of the chain. Both NEXT commands move forward to the first instance in the appropriate segment:

```
REPOSITION Emp_ID;
NEXT Emp_ID;
NEXT Pay_Date;
```

If one of the NEXT commands uses the FOR prefix, it works the same as described above, but rather than moving forward only one segment instance, NEXT moves forward however many records the FOR specifies. For example, the following retrieves the first instance in the EmpInfo segment and the next three instances of the SalInfo segment. All three records are for only one employee because the first NEXT establishes the position:

```
REPOSITION Emp_ID;
NEXT Emp_ID;
FOR 3 NEXT Pay_Date INTO Stemp;
```

After this code is executed, the stack contains data from the following segments:

1. Emp_ID instance 1 and Pay_Date instance 1
2. Emp_ID instance 1 and Pay_Date instance 2
3. Emp_ID instance 1 and Pay_Date instance 3

Every NEXT command that uses a FOR prefix does so independent of any other NEXT command. If there are two NEXT commands, the first executes. When it is complete, the position is the last instance retrieved. The second NEXT command then executes and retrieves data from within the parent established by the first NEXT. In the following example, the first NEXT retrieves the first two instances from the EmpInfo segment and places the instances into the stack. The second NEXT retrieves the next three instances of the SalInfo segment. Note its parent instance is the second EmpInfo segment instance. The stack variable FocCount indicates the number of rows currently in the stack. The prefix Stemp is needed to indicate the stack.

STACK CLEAR Stemp;
REPOSITION Emp_ID;
FOR 2 NEXT Emp_ID INTO Stemp(1);
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount);

The stack contains data from the following segments after the first NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2

The stack contains data from the following segments after the second NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2 and Pay_Date instance 1
3. Emp_ID instance 2 and Pay_Date instance 2
4. Emp_ID instance 2 and Pay_Date instance 3

The row in the INTO stack that the output is placed in is specified by supplying the row number after the stack name. When two NEXT commands are used in a row for the same stack, care must be taken to ensure that data is written to the appropriate row in the stack. If a stack row number is not specified for the second NEXT command, data is placed into the last row written to by the first NEXT, and existing data is overwritten. In order to place data in a different row, a row number or an expression to calculate the row number can be used. For example, the second NEXT command specifies the row after the last row by adding one to the variable FocCount:

FOR 2 NEXT Emp_ID INTO Stemp(1);
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount+1);

The stack now appears as follows. Notice that there is a new row 2:

1. Emp_ID instance 1
2. Emp_ID instance 2
3. Emp_ID instance 2 and Pay_Date instance 1
4. Emp_ID instance 2 and Pay_Date instance 2
5. Emp_ID instance 2 and Pay_Date instance 3

If a WHERE phrase is specified, the NEXT moves forward as many times as necessary to retrieve one record that satisfies the selection criteria specified in the WHERE phrase. For instance, the following retrieves the next record where the Gross field of the child segment is greater than 1,000. Like the previous example, the data retrieved is only for the employee that the first NEXT retrieves:

```plaintext
NEXT Emp_ID;
NEXT Pay_Date WHERE Gross GT 1000;
```

If both a FOR prefix and a WHERE phrase are specified, the NEXT moves forward as many times as necessary to retrieve the number of records specified in the FOR prefix that satisfy the selection criteria specified in the WHERE phrase.

For example, the following syntax retrieves the next three records where the Gross field of the child segment is greater than 1,000. As above, the data retrieved is only for the employee that the first NEXT retrieves:

```plaintext
NEXT Emp_ID;
FOR 3 NEXT Pay_Date INTO Stemp WHERE Gross GT 1000;
```

**Unique Segments**

Maintain allows separate segments to be joined in a one-to-one relation (among other ways). Unique segments are indicated by specifying a SEGTYPE of U, KU or DKU in the Master File, or by issuing a JOIN command. In a NEXT command, you retrieve a unique segment by specifying a field from the segment in the field list of the command. You cannot specify the unique segment as an anchor segment.

If an attempt is made to retrieve data from a unique segment and the segment does not exist, the fields are treated as if they are fields in the parent segment. This means that the returned data is spaces, zeroes, and/or nulls (missing values), depending on how the segment is defined. In addition, the answer set contains as many rows as the parent of the unique segment. If an UPDATE or a DELETE command subsequently uses the data in the stack and the unique segment does not exist, it is not an error, because unique segments are treated as if the fields are fields in the parent. If an INCLUDE is issued, the data source is not updated.
ON MATCH

The ON MATCH command defines the action to take if the prior MATCH command succeeds (if it is able to retrieve the specified segment instance). There can be intervening commands between the MATCH and ON MATCH commands, and they can be in separate functions.

You should query the FocFetch system variable in place of issuing the ON MATCH command. FocFetch accomplishes the same thing more efficiently. For more information, see FocFetch on page 110.

Syntax: How to Use the ON MATCH Command

The syntax of the ON MATCH command is

```
ON MATCH command
```

where:

```
command
```

Is the action that is taken when the prior MATCH command succeeds.

You can specify any Maintain command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

Example: Using On MATCH

The following example displays a message stating that there was a MATCH:

```
MATCH Emp_ID;
ON MATCH TYPE "Employee was found in the data source";
```

This example shows an UPDATE that is performed after a MATCH occurs:

```
MATCH Emp_ID;
ON MATCH UPDATE Salary FROM SalStack;
```

The following shows several commands being executed after a MATCH:

```
MATCH Emp_ID;
ON MATCH BEGIN
    TYPE "Employee was found in the data source";
    UPDATE Salary FROM SalStack;
    PERFORM Jobs;
ENDBEGIN
```
ON NEXT

The ON NEXT command defines the action to take if the prior NEXT command succeeds (if it is able to retrieve all of the specified records). There can be intervening commands between the NEXT and ON NEXT commands, and they can be in separate functions.

It is recommended that you query the FocFetch system variable in place of issuing the ON NEXT command. FocFetch accomplishes the same thing more efficiently. For more information, see *FocFetch* on page 110.

**Syntax:** How to Use the ON NEXT Command

The syntax of the ON NEXT command is

```
ON NEXT command
```

where:

`command`

Is the action that is taken when NEXT is successful.

You can specify any Maintain command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

**Example:** Using ON NEXT

The first example displays a message stating that the NEXT was successful:

```
NEXT Emp_ID;
ON NEXT TYPE "Was able to NEXT another employee";
```

This example computes a five percent increase for the next employee in the data source:

```
NEXT Emp_ID;
ON NEXT COMPUTE NewSal/D12.2 = Curr_Sal * 1.05;
```

The following example shows several commands that are executed after a NEXT:

```
ON NEXT BEGIN
  TYPE "Was able to NEXT another employee";
  COMPUTE NewSal/D12.2 = Curr_Sal * 1.05;
ENDBEGIN
```

ON NOMATCH

The ON NOMATCH command defines the action to take if the prior MATCH command fails (if it is unable to retrieve the specified segment instance). There can be intervening commands between the MATCH and ON NOMATCH commands, and they can be in separate functions.
It is recommended that you query the FocFetch system variable in place of issuing the ON NOMATCH command. FocFetch accomplishes the same thing more efficiently. For more information, see *FocFetch* on page 110.

**Syntax:**

**How to Use the ON NOMATCH Command**

The syntax of the ON NOMATCH command is

```
ON NOMATCH command
```

where:

- `command`
  - Is the action that is taken when the prior MATCH command fails.
  - You can specify any Maintain command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

**Example:**

**Using ON NOMATCH**

The first example displays a message stating that the MATCH was unsuccessful:

```
MATCH Emp_ID;
ON NOMATCH TYPE "Employee was not found in the data source";
```

This example shows an INCLUDE of a row from the Emp stack:

```
MATCH Emp_ID FROM Emp(Cnt);
ON NOMATCH INCLUDE Emp_ID FROM Emp(Cnt);
```

The following example shows several commands that are executed after a MATCH command fails:

```
MATCH Emp_ID;
ON NOMATCH BEGIN
  TYPE "Employee was not found in the data source";
  INCLUDE Emp_ID;
  PERFORM Cleanup;
ENDBEGIN
```

**ON NONEXT**

The ON NONEXT command defines the action to take if the prior NEXT command fails (if it is unable to retrieve all of the specified records). There can be intervening commands between the NEXT and ON NONEXT commands, and they can be in separate functions.

For example, when the following NEXT command is executed

```
FOR 10 NEXT Emp_ID INTO Stkemp;
```
only eight employees are left in the data source, so only eight records are retrieved, raising
the ON NONEXT condition.

It is recommended that you query the FocFetch system variable in place of issuing the ON
NONEXT command. FocFetch accomplishes the same thing more efficiently. For more
information, see FocFetch on page 110.

Syntax:  How to Use the ON NONEXT Command

The syntax of the ON NONEXT command is

    ON NONEXT command

where:

command

    Is the action that is taken when NEXT fails.

    You can specify any Maintain command except for CASE, DECLARE, DESCRIBE, END,
    MAINTAIN, MODULE, and another ON command.

Example:  Using ON NONEXT

The first example displays a message stating that the NEXT was unsuccessful:

    NEXT Emp_ID;
    ON NONEXT TYPE "There are no more employees";

If all of the employees have been processed, the program is exited:

    NEXT Emp_ID;
    ON NONEXT GOTO EXIT;

The following example shows several commands being executed after a NEXT fails:

    ON NONEXT BEGIN
        TYPE "There are no more employees in the data source";
        PERFORM Wrapup;
    ENDBEGIN

PERFORM

    You can use the PERFORM command to pass control to a Maintain function. Once that function
    is executed, control returns to the command immediately following the PERFORM.
Syntax: How to Use the PERFORM Command

The syntax of the PERFORM command is

PERFORM functionname [()];

where:

functionname
   Specifies the name of the function to perform.

()  
   Is optional. If you omit the word PERFORM and just use the function name, parentheses are required.

;  
   Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.

For example, to perform the function named NextSet, issue the command:

PERFORM NextSet;

Reference: Commands Related to PERFORM

- CASE /ENDCASE defines a Maintain function.
- GOTO transfers control to another function or to the end of the current function.

Using PERFORM to Call Maintain Functions

When you call a function as a separate statement (that is, outside of a larger expression), if the preceding command can take an optional semicolon terminator but was coded without one, you must call the function in a COMPUTE or PERFORM command. (You can use PERFORM for Maintain functions only, though not for Maintain functions that return a value.)

For example, in the following source code, the NEXT command is not terminated with a semicolon, so the function that follows it must be called in a PERFORM command:

NEXT CustID INTO CustStack
PERFORM VerifyCustID();
However, in all other situations, you can call functions directly, without a PERFORM command. For example, in the following source code, the NEXT command is terminated with a semicolon, so the function that follows it can be called without a PERFORM command:

```
NEXT CustID INTO CustStack;
VerifyCustID();
```

**Note:** When calling a function without using a PERFORM command, you must include parentheses.

For more information about terminating commands with semicolons, see *Terminating Command Syntax* on page 24.

**Using PERFORM With Data Source Commands**

A PERFORM can be executed in a MATCH command following an ON MATCH or ON NOMATCH command, or in NEXT following ON NEXT or ON NONEXT. In the following example, the function NotHere is performed after a NOMATCH condition occurs:

```
ON NOMATCH PERFORM NotHere;
```

**Nesting PERFORM Commands**

PERFORM commands can branch to functions containing other PERFORM commands. As each ENDCASE command is encountered, control returns to the command after the most recently executed PERFORM command. In this manner, control eventually returns to the original PERFORM.

**Avoiding GOTO With PERFORM**

It is recommended that you do not include a GOTO command within the scope of a PERFORM command. See *GOTO* on page 112 for information on the incompatibility of the PERFORM and GOTO commands.

**REPEAT**

The REPEAT command enables you to loop through a block of code. REPEAT defines the beginning of the block, and ENDREPEAT defines the end. You control the loop by specifying the number of loop iterations, and/or the conditions under which the loop terminates. You can also define counters to control processing within the loop, for example incrementing a row counter to loop through the rows of a stack.
How to Use the REPEAT Command

The syntax of the REPEAT command is:

```
REPEAT {int|ALL|WHILE condition|UNTIL condition} [counter [/fmt] = init_expr;] [;]
command
...
ENDREPEAT [counter[/fmt]=increment_expr;...]
```

where:

`int`
Specifies the number of times the REPEAT loop is to run. The value of `int` can be an integer constant, an integer field, or a more complex expression that resolves to an integer value. If you use an expression, the expression should resolve to an integer, although other types of expressions are possible. If the expression resolves to a floating-point or packed-decimal value, the decimal portion of the value will be truncated. If it resolves to a character representation of a numeric value, it will be converted to an integer value.

Expressions are described in Expressions Reference on page 27.

`ALL`
Specifies that the loop is to repeat indefinitely, terminating only when a GOTO EXITREPEAT command is issued from within the loop.

`WHILE`
Specifies that the WHILE condition is to be evaluated prior to each execution of the loop. If the condition is true, the loop is entered. If the condition is false, the loop is terminated and control passes directly to the command immediately following ENDREPEAT. If the condition is false when the REPEAT command is first executed, the loop is never entered.

`UNTIL`
Specifies that the UNTIL condition is to be evaluated prior to each execution of the loop. If the condition is false, the loop is entered. If the condition is true, the loop is terminated and control passes directly to the command immediately following ENDREPEAT. If the condition is true when the REPEAT command is first executed, the loop is never entered.

`condition`
Is a valid Maintain expression that can be evaluated as true or false (that is, a Boolean expression).
counter
Is a variable that you can use as a counter within the loop. You can assign the initial value of the counter in the REPEAT command, or in a COMPUTE command issued prior to the REPEAT command. You can increment the counter at the end of each loop iteration in the ENDREPEAT command. If you wish, you can also change the value of the counter in a COMPUTE command within the loop. You can refer the counter throughout the loop, including:

- Inside the loop, as a stack subscript.
- Inside the loop, in an expression.
- In a WHILE or UNTIL condition in the REPEAT command.

fmt
Is the format of the counter. It can be any valid format except for TX. The format is required only if you are defining the variable in this command.

initExpr
Is an expression whose value is assigned to the counter before the first iteration of the loop. It can be any valid Maintain expression.

incrementExpr
Is an expression whose value is assigned to the counter at the end of each complete loop iteration. It can be any valid Maintain expression.

command
Is one or more Maintain commands, except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE.


Terminates the command. If you do not specify a counter, the semicolon is optional but recommended. Including it allows for flexible syntax and better processing. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.
**Example: Simple Loops**

The following code has a loop that executes ten or fewer times. The REPEAT line initiates the loop. The number 10 indicates that the loop will run ten times, barring any conditions or commands to exit the loop. The ON NONEXT GOTO EXITREPEAT command causes the loop to be exited when there are no more instances of Sales in the data source. The COMPUTE command calculates TotSales within an execution of the loop. The ENDREPEAT command is the boundary for the loop. Commands after ENDREPEAT are not part of the loop.

Because there is no loop counter, there is no way to know which repetition of the loop is currently executing:

```plaintext
COMPUTE TotSales = 0;
REPEAT 10;
   NEXT Sales;
   ON NONEXT GOTO EXITREPEAT;
   COMPUTE TotSales = TotSales + Sales;
ENDREPEAT
```

**Example: Specifying Loop Iterations**

You can control the number of times that the flow of control cycles through the loop by specifying the number of iterations. For example:

```plaintext
REPEAT 27;
```

You can also specify a condition that must be true or false for looping to continue:

```plaintext
REPEAT WHILE Rows GT 15;
```

**Example: Repeating a Loop a Variable Number of Times**

The REPEAT variable construct indicates that the loop is repeated the number of times indicated by the value of the variable. In this example, Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack. The loop executes a variable number of times based on the value of Stk1.FocCount:

```plaintext
FOR ALL NEXT Country INTO Stk1;
COMPUTE Cnt/I4 = 1;
REPEAT Stk1.FocCount;
   TYPE "Country = <Stk1(Cnt).Country";
   COMPUTE Cnt = Cnt + 1;
ENDREPEAT
```
Example: REPEAT WHILE and UNTIL

The REPEAT WHILE construct indicates that the loop should be repeated as long as the expression is true. Once the expression is no longer true, the loop is exited. In this example, the loop will be executed Stk1.FocCount number of times. Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack:

FOR ALL NEXT Country INTO Stk1;
COMPUTE Cnt/I4 = 1;
REPEAT WHILE Cnt LE Stk1.FocCount;
    TYPE "Country = <Stk1(Cnt).Country ";
    COMPUTE Cnt = Cnt + 1;
ENDREPEAT

The REPEAT UNTIL construct indicates that the loop is repeated as long as the expression is not true. Once the expression is true, the loop is exited. In this example, the loop is executed Stk1.FocCount number of times. Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack. The COMPUTE increments the counter, although this could have been specified on the ENDREPEAT command. ENDREPEAT indicates the end of the loop:

FOR ALL NEXT Country INTO Stk1;
COMPUTE Cnt/I4 = 1;
REPEAT UNTIL Cnt GT Stk1.FocCount;
    TYPE "Country = <Stk1(Cnt).Country ";
    COMPUTE Cnt = Cnt + 1;
ENDREPEAT

Example: Establishing Counters

You can use as many counters as you wish in each loop. The only restriction is that all counter initializations performed in the REPEAT command must fit on the single line of the REPEAT command, and all counter incrementation performed in the ENDREPEAT command must fit on the single line of the ENDREPEAT command. You can avoid the single-line limitation by defining and incrementing counters using COMPUTE commands. It is legitimate, however, to have a REPEAT loop and never refer to any counter within the loop. If this is done, the same row of data is always worked on and unexpected results can occur.
The following examples do not have any index notation on the stack Stkemp, so each NEXT puts data into the same row of the stack. In other words, INTO Stkemp is the same as INTO Stkemp(1). Row one is always referenced because, by default, if there is a stack name without a row number, the default row number of one is used.

```
REPEAT 10;
  NEXT Emp_ID INTO Stkemp;
  .
  .
ENDREPEAT
```

is the same as:

```
REPEAT 10 Cnt/I4=1;
  NEXT Emp_ID INTO Stkemp;
  .
  .
ENDREPEAT Cnt=Cnt+1;
```

To resolve this problem, the REPEAT loop can establish counters and how they are incremented. Inside the loop, individual rows of a stack can be referenced using one of the REPEAT loop counters. The REPEAT command can be used to initialize many variables that will be used in the loop. For example

```
REPEAT 100 Cnt/I4=1; Flag=IF Factor GT 10 THEN 2 ELSE 1;
```

or:

```
REPEAT ALL Cnt = IF Factor GT 10 THEN 1 ELSE 10;
```

On the ENDREPEAT command the counters are incremented by whatever calculations follow the keyword ENDREPEAT. Two examples are

```
ENDREPEAT Cnt = Cnt + 1; Flag = Flag*2;
```

and:

```
ENDREPEAT Cnt=IF Department EQ 'MIS' THEN Cnt+5 ELSE Cnt+1;
```
The following code sets up a repeat loop and computes the variable New_Sal for every row in the stack. The REPEAT line initiates the loop. The ALL indicates that the loop continues until a command in the loop tells the loop to exit. A GOTO EXITREPEAT command is needed in a loop when REPEAT ALL is used. The Cnt = 1 initializes the counter to 1 the first time through the loop. The COMPUTE command calculates a five percent raise. It uses the REPEAT counter (Cnt) to access each row in the stack one at a time. The counter is checked to see if it is greater than or equal to the number of rows in the Stkemp stack. The stack variable FocCount always contains the value of the number of rows in the stack. After every row is processed, the loop is exited.

The ENDREPEAT command contains the instructions for how to increment the counter:

```
REPEAT ALL Cnt/I4=1;
   COMPUTE Stkemp(Cnt).NewSal/D12.2=Stkemp(Cnt).Curr_Sal * 1.05;
   IF Cnt GE Stkemp.FocCount THEN GOTO EXITREPEAT;
ENDREPEAT Cnt=Cnt+1;
```

**Example: Nested REPEAT Loops**

REPEAT loops can be nested. This example shows one repeat loop nested within another loop. The first REPEAT command indicates that the loop will run as long as the value of A is less than 3. It also initializes the counter A to 1. The second REPEAT command indicates that the nested loop will run until the value of B is greater than 4. It initializes the counter B to 1. Two ENDREPEAT commands are needed, one for each REPEAT command. Each ENDREPEAT increments its respective counters.

```
REPEAT WHILE A LT 3; A/I4 = 1;
   TYPE "In A loop with A = <A";
   REPEAT UNTIL B GT 4; B/I4 = 1;
      TYPE " ***In B loop with B = <B ";
   ENDREPEAT B = B + 1;
ENDREPEAT A = A + 1;
```

The output of these REPEAT loops would look like the following:

```
In A loop with A = 1
   ***In B loop with B = 1
   ***In B loop with B = 2
   ***In B loop with B = 3
   ***In B loop with B = 4
In A loop with A = 2
   ***In B loop with B = 1
   ***In B loop with B = 2
   ***In B loop with B = 3
   ***In B loop with B = 4
```
**Reference:** Usage Notes for REPEAT

The actual number of loop iterations can be affected by other phrases and commands in the loop. The loop can end before completing the specified number of iterations if it is terminated by a WHERE or UNTIL condition, or by a GOTO EXITREPEAT command issued within the loop.

**Reference:** Commands Related to REPEAT

- **COMPUTE** is used to define user-defined variables and assign values to existing variables.
- **GOTO** transfers control to another function or to the end of the current function.

**Branching Within a Loop**

There are two branching instructions that facilitate the usage of REPEAT and ENDREPEAT to control loop iterations:

- **GOTO ENDREPEAT** causes a branch to the end of the repeat loop and executes any computes on the ENDREPEAT line.
- **GOTO EXITREPEAT** causes the loop to be exited and goes to the next logical instruction after the ENDREPEAT.

**Example:** Terminating the Loop From the Inside

You can terminate a REPEAT loop by branching from within the loop to outside the loop. When you issue the command GOTO EXITREPEAT, Maintain branches to the command immediately following the ENDREPEAT command. It does not increment counters specified in the ENDREPEAT command. For example:

```plaintext
REPEAT ALL;
.
.
  GOTO EXITREPEAT;
.
.
ENDREPEAT
```

**REPOSITION**

For a specified segment and each of its descendants, the REPOSITION command resets the current position to the beginning of chain for that segment. That is, each segment is reset to just prior to the first instance.
Most data source commands change the current segment position to the instance that they most recently accessed. When you wish to search an entire data source or path for records, start at the beginning of the data source or path by first issuing the REPOSITION command.

**Syntax:** How to Use the REPOSITION Command

The syntax of the REPOSITION command is

```
REPOSITION segment_spec [;
```

where:

- `segment_spec` - Is the name of a segment or the name of a field in a segment. The specified segment and all of its descendants are repositioned to the beginning of the segment chain.
- `;` - Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.

**Example:** Using REPOSITION

The following example repositions the root segment and all of the descendant segments of the Employee data source:

```
REPOSITION Emp_ID;
```

The next example repositions both the SalInfo and Deduct segments in the Employee data source:

```
REPOSITION Pay_Date;
```

**Reference:** Commands Related to REPOSITION

- **NEXT** starts at the current position and moves forward through the data source and can retrieve data from one or more records.
- **MATCH** searches entire segments for a matching field value and can retrieve an exact match in the data source.

**REVISE**

The REVISE command reads a stack of transaction data and writes it to a data source, inserting new segment instances and updating existing instances.
REVISE combines the functionality of the INCLUDE and UPDATE commands. It reads each stack row and processes each segment in the specified path using the following logic:

MATCH key
ON MATCH UPDATE fields
ON NOMATCH INCLUDE segment

You specify a path running from an anchor segment to a target segment. For each segment in the path, REVISE matches the instance of the segment in the stack against the corresponding instances in the data source. If the keys of an instance fail to find a match in the data source, REVISE adds the instance. If an instance does find a match, REVISE updates it using the fields that you have specified. The values that REVISE writes to the data source are provided by the stack.

Data source commands treat a unique segment as an extension of its parent, so that the unique fields seem to reside in the parent. Therefore, when REVISE adds an instance to a segment that has a unique child, it automatically also adds an instance of the child.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This enables REVISE to navigate from the root to the first instance of the anchor segment.

**Syntax:**

**How to Use the REVISE Command**

The syntax of the REVISE command is

```plaintext
[FOR {int|ALL} REVISE data_spec [FROM stack [(row)]] [;]
```

where:

**FOR**

Indicates that an integer or ALL will be used to specify how many stack rows to write to the data source.

If you specify FOR, you must also specify a source stack using the FROM phrase. If you omit FOR, REVISE defaults to writing one row.

**int**

Is an integer expression that specifies the number of stack rows to write to the data source.

**ALL**

Specifies that all of the rows of the stack are to be written to the data source.
**data_spec**

Identifies the path to be written to the data source and the fields to be updated:

1. Specify each field that you want to update in existing segment instances. You can update only non-key fields. Because a key uniquely identifies an instance, keys can be added and deleted but not changed.

2. Specify the path by identifying its anchor and target segments. You can specify a segment by providing its name or the name of one of its non-key fields.

If you have already identified the anchor and target segments in the process of specifying update fields, you do not need to do anything further to specify the path. Otherwise, if either the anchor or the target segment has not been identified using update fields, specify it using its segment name.

**FROM**

Indicates that the transaction data will be supplied by a stack. If this is omitted, the transaction data is supplied by the Current Area.

**stack**

Is the name of the stack whose data is being written to the data source.

**row**

Is a subscript that specifies the first stack row to be written to the data source. If omitted, it defaults to 1.

### Example: Using REVISE

In the following example the user is able to enter information for a new employee, or change the last name of an existing employee. Existing employee records are displayed in a grid. All of the information is stored in a stack named EmpStk.

```
MAINTAIN FILE EMPLOYEE
FOR ALL NEXT Emp_ID INTO EmpStk;
Wiform Show GetData;
CASE Alter_Data
FOR ALL REVISE Last_Name FROM EmpStk;
ENDCASE
END
```
When the function Alter_Data is called from an event handler of a form, the REVISE command reads EmpStk and tries to find the Emp_ID of each row in the Employee data source. If Emp_ID exists in the data source, REVISE updates the Last_Name field of that segment instance. If it does not exist, then REVISE inserts a new EmpInfo instance into the data source, and writes the fields of EmpInfo from the stack to the new instance.

**Reference:** Usage Notes for REVISE

Maintain requires that the data sources to which it writes have unique keys.

**Reference:** Commands Related to REVISE

- **INCLUDE** adds new segment instances to a data source.
- **UPDATE** updates data source fields.
- **COMMIT** makes all data source changes since the last COMMIT permanent.
- **ROLLBACK** cancels all data source changes made since the last COMMIT.

**ROLLBACK**

The ROLLBACK command processes a logical transaction. A logical transaction is a group of data source changes that are treated as one. The ROLLBACK command cancels prior UPDATE, INCLUDE, and DELETE operations that have not yet been committed to the data source using the COMMIT command.

**Syntax:** How to Use the ROLLBACK Command

The syntax of the ROLLBACK command is

```
ROLLBACK [;]
```

where:

```
;
```

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.
Example: Using ROLLBACK

This example shows part of a procedure where an employee ID needs to be changed. Because Emp_ID is a key, it cannot be changed. To accomplish this, it is necessary to collect the existing field values, make the necessary changes, delete the employee from the data source, and add a new segment instance.

The following shows partial code where the existing instance is deleted and a new one is added. If for some reason the INCLUDE does not work, the DELETE should not occur.

```sql
CASE Chngempid
  DELETE Emp_ID;
  IF FocError NE 0 PERFORM DeleteError;
  INCLUDE Emp_ID Bank_Name Dat_Inc Type Pay_Date Ded_Code;
  IF FocError NE 0 PERFORM Undo;
ENDCASE

CASE Undo
  ROLLBACK;
ENDCASE
```

Reference: Usage Notes for ROLLBACK

- A ROLLBACK is automatically issued when a program is exited abnormally.
- A successful ROLLBACK issued in a called procedure frees the data source position maintained by that procedure and by all calling procedures.
- A ROLLBACK is automatically issued if an attempt to COMMIT fails.

DBMS Combinations

When an application accesses more than one DBMS (for example, FOCUS and Teradata), ROLLBACK is treated as a broadcast rollback. There is no coordination between the different types of data sources, therefore the ROLLBACK might succeed against one type of data source but fail against another.

SAY

The SAY command writes messages to a file or to the default output device. You can use SAY for application debugging, such as tracing application control flow, and for recording an accounting trail. If you wish to display messages to application users, you should use forms, which provide superior display capabilities and better control than the SAY command.
How to Use the SAY Command

The syntax of the SAY command is

```
SAY [TO ddname] expression [expression ...] ;
```

where:

- **expression**
  - Is any Maintain expression. Multiple expressions must be separated by spaces.
  - Each message is written on the current line, beginning in the column that follows the end of the previous message. When a message reaches the end of the current line in the file or display device, or encounters a line feed (the string \n) in the message text, the message stream continues in column 1 of the next line.
  - If you write to output devices that buffer messages before displaying them, you may wish to end each message with an explicit line feed to force the buffer to display the last line of the message.

- **TO ddname**
  - Specifies the logical name of the file to which the SAY message is written. *ddname* is a character expression. If you supply a literal for *ddname*, it must be enclosed by single or double quotation marks.
  - You must define the logical name using a DYNAM command on z/OS before the SAY command is executed. In order to append to an existing file (for example, to write to a file from more than one procedure), specify the appropriate option in the DYNAM command.
  - If *TO ddname* is omitted, the message is written to the default output device of the environment in which the SAY command is issued. For example:
    - In a web-deployed application, the message is written to the webpage currently displayed in the web browser.
      - If the message is written from a procedure that resides on a different server than the webpage, the message is prefixed with "(FOC03764) From Server ==> " to indicate that it was posted by a remote procedure.
    - In a Windows-deployed application run from the Maintain Development Environment, the message is written to the Run tab of the output window.
  - In addition, if *TO ddname* is omitted and this procedure was called remotely (that is, called using a CALL *procname* AT command), the message will also be copied to the FocMsg stack of the calling procedure.
Note: Literals must be enclosed in single or double quotation marks, while variables do not appear in quotes.

Reference: Commands Related to SAY

TYPE writes messages to a file or to a form.

Writing Segment and Stack Values

You can use the SEG and STACK prefixes to write the values of all the fields of a segment or columns of a stack to a message. This can be helpful when you write messages to log and checkpoint files.

SEG.fieldname inserts Current Area values for all of the fields of the specified segment.
STACK.stackname(row) inserts, for the specified stack, the values of the specified row.

Choosing Between the SAY and TYPE Commands

The rules for specifying messages using the SAY command are simpler and more powerful than those for the TYPE command. For example, you can include all kinds of expressions in a SAY command, but you can only include character string constants and scalar variables in a TYPE command.

Note that, unlike the TYPE command, the SAY command does not generate a default line feed at the end of each line.

SET

You can change parameters that control output, work areas, and other features in your WebFOCUS Server environment by using the SET command from a WebFOCUS procedure (not a Maintain procedure).

This command is outside the Maintain language, but is described here for your convenience, since many of these settings affect how WebFOCUS Maintain behaves.

However, you can change a limited number of SET parameters from within a Maintain procedure using SYSMGR.FOCSET. For more information, see SYS_MGR.FOCSET on page 184.

For a list of SET parameters, see Customizing Your Environment in the Developing Reporting Applications manual.
**Syntax:** How to Use SET Parameters

The syntax is

```
SET parameter = option[, parameter = option,...]
```

where:

- `parameter`
  - Is the setting you wish to change.

- `option`
  - Is one of a number of options available for each parameter.

You can set several parameters in one command by separating each with a comma.

You may include as many parameters as you can fit on one line. Repeat the SET keyword for each new line.

**Note:** This syntax is valid *only* in a WebFOCUS procedure.

**Syntax:** How to Use SET Parameters in a Request

Many SET commands that change system defaults can be issued from within FOCUS TABLE requests. SET used in this manner is temporary, affecting only the current request.

The syntax is

```
ON TABLE SET parameter value [AND parameter value ...]
```

where:

- `parameter`
  - Is the system default you wish to change.

- `value`
  - Is an acceptable value that will replace the default value.

**Note:** This syntax is valid *only* in a WebFOCUS report procedure.

**SetLayer**

The SetLayer command allows layers to be set as visible or invisible at run time. It can be used in a Maintain or a JavaScript event.
**Syntax:** How to Use the SetLayer Command

The syntax of the SetLayer command is

**Maintain**

```
Formname.SetLayer.("layername",n);
```

**JavaScript**

```
setLayer.formname.("layername",n);
```

**Note:** JavaScript is case-sensitive.

where:

- "layername"
  - Is the name of the layer, enclosed in double quotation marks.

- n
  - Possible values are:
    - 0 sets the layer to invisible (off).
    - 1 sets the layer to visible (on).

To preset a layer before the form is run, you can use the SetLayer command with the Winform Show_inactive command:

```
Winform Show_inactive Form1;
Form1.SetLayer("Layer1",1);
Form1.SetLayer("Layer2",0);
Winform Show Form1;
```

This sets Layer1 as visible and Layer2 as invisible.

### STACK CLEAR

STACK CLEAR clears the contents of each of the stacks listed, so that each stack has no rows. This sets the FocCount variable to zero and FocIndex variable to one for each stack.

**Syntax:** How to Use the STACK CLEAR Command

The syntax of the STACK CLEAR command is

```
STACK CLEAR stacklist [;]
```
where:

stacklist

Specifies the stacks to be initialized. Stack names are separated by blanks.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see Terminating Command Syntax on page 24.

**Example:** Using STACK CLEAR

The following initializes the Emp stack:

STACK CLEAR Emp;

The next example initializes both the Emp and the Dept stacks:

STACK CLEAR Emp Dept;

**STACK SORT**

The STACK SORT command enables you to sort the contents of a stack in ascending or descending order based on the values in one or more columns.

**Syntax:** How to Use the STACK SORT Command

The syntax for the STACK SORT command is

STACK SORT stackname BY [HIGHEST] column [BY [HIGHEST] column ...] [;]

where:

stackname

Specifies the stack to be sorted. The stack name cannot be subscripted with a row range in order to sort only part of the stack.

HIGHEST

If specified, sorts the stack in descending order. If not specified, the stack is sorted in ascending order.

column

Is a stack column name. At least one column name must be specified. The column must exist in the specified stack.
Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.

**Example: Using STACK SORT**

The following sorts the stack Emp using the values in the stack column Emp_ID:

```sql
STACK SORT Emp BY Emp_ID;
```

The following sorts the stack Emp so that the employees with the highest Salary are placed first in the stack:

```sql
STACK SORT Emp BY HIGHEST Salary;
```

The next example sorts the stack by Department. Within Department, the rows are ordered by highest Salary:

```sql
STACK SORT Emp BY Department BY HIGHEST Salary;
```

**Sorting Data With the Using CASE_INSENSITIVE Parameter**

You can sort data in a stack without considering case by implementing the using CASE_INSENSITIVE parameter.

**Syntax: How to Sort Data in a Stack With the Using CASE_INSENSITIVE Parameter**

To sort data in a stack and ignore case-sensitivity:

1. Add the `using CASE_INSENSITIVE` tablename parameter after the BY fields:

   ```sql
   STACK SORT stackname BY [HIGHEST] column [BY [HIGHEST] column ...] [using CASE_INSENSITIVE] [;]
   ``'

2. Import the MNTUWS function library.

3. Add the following command to the top case of the Maintain procedure:

   ```sql
   perform prep_CASE_INSENSITIVE
   ```
For example:

```maintain
MAINTAIN
module import (mntuws)
$ Declarations
case top
perform prep_CASE_INSENSITIVE;
COMPUTE STK(1).NAME/A10="GEORGE"
STK(2).NAME="Bernard"
STK(3).NAME="Shaw"
...
STACK SORT STK BY NAME using CASE_INSENSITIVE;
...
```

**Note:** For a back-end procedure without a top case, the `perform prep_CASE_INSENSITIVE` command is not required.

When using MNTCON MATCH_CASE ON, the case of the code must be the same as it appears above.

**SYS_MGR**

The SYS_MGR global object provides functions and variables that control the environment for your WebFOCUS Maintain application. It can help developers ensure the most efficient interaction between Maintain and DBMS servers, as well as manage WebFOCUS environment and run-time variables.

For use with relational data sources, SYS_MGR can be used from within Maintain procedures to:

- Deactivate preliminary database operation checking by Maintain before an update and rely on the DBMS to perform its own internal checking, thus reducing processing time and resources (SYS_MGR.PRE_MATCH).

- Retrieve DBMS error codes, allowing developers to code applications to efficiently recover from error conditions (SYS_MGR.DBMS_ERRORCODE).

- Issue native SQL commands directly from a Maintain procedure (SYS_MGR.ENGINE).

The SYS_MGR syntax can also be used from within a Maintain procedure to:

- Issue system settings for WebFOCUS servers on the fly (SYS_MGR.FOCSET).

- Set values for Userid and Password for target servers before issuing an EXEC AT or CALL AT.
SYS_MGR.DBMS_ERRORCODE

 Retrieve input parameter values passed at invocation time
 (SYS_MGR.GET_INPUTPARAMS_COUNT, SYS_MGR.GET_NAMEPARM,
  SYS_MGR.GET_POSITIONPARM).

 Note: All sys_mgr function and variable names are case-insensitive. You can use
 SYS_MGR.DBMS_ERRORCODE, sys_mgr.dbms_errorcode, sys_mgr.DBMS_ErrorCode, and
 so on, interchangeably.

SYS_MGR.DBMS_ERRORCODE

 The SYS_MGR.DBMS_ERRORCODE syntax enables you to retrieve error codes returned by the
 DBMS server and then take appropriate action. For example, a developer might want to take
 a different course of action for an INSERT that fails because the user does not have INSERT
 rights versus a referential integrity failure.

 Note:  

 The return codes are DBMS specific. The DB2 return codes do not match the Oracle
 code. Moreover, DBMS vendors have been known to change the return codes on release
 boundaries. You should clearly document that you are using this feature so sufficient
 testing can be done before rolling in a new DBMS.

 DBMS_ERRORCODE is local to the current Maintain procedure.

 Syntax: How to Use SYS_MGR.DBMS_ERRORCODE

 The syntax is

 SYS_MGR.DBMS_ERRORCODE ;

 Example: Retrieving an Error Code From a DBMS

 For example, the following Maintain code will retrieve an error code from a DBMS, and if it is
 a specific code, branches to some appropriate code:

 Compute ErrCode/a3 = SYS_MGR.DBMS_ERRORCODE ;
 If ErrCode EQ '515' goto BadInsert;

 SYS_MGR.ENGINE

 You can issue DBMS commands directly (SQL Passthru) from a Maintain procedure using the
 SYS_MGR.ENGINE command.
**Note:** Problems with direct commands are not reported in FOCERROR. You will need to use DBMS_ERRORCODE to determine the success or failure of these commands.

**Syntax:**  
How to Use the SYS_MGR.ENGINE Command

The syntax for the SYS_MGR.ENGINE command is

```
SYS_MGR.ENGINE("enginename", "command");
```

where:

- **enginename**
  - Is the name of the RDBMS to which you are passing the command. For a complete list of the possible values, see the *Adapter Administration for UNIX, Windows, OpenVMS, IBM i, and z/OS* manual.

- **command**
  - Is any valid SQL command, including CREATE, DROP, and INSERT.

**Example:**  
Issuing the DROP TABLE Command

The following command drops the table NYACCTS. The error code is saved in a variable named rc.

```
Compute rc/i8;
rc = sys_mgr.engine("SQLMSS", "DROP TABLE NYACCTS");
Type "Return Code=<<rc  DBMS Err=<<SYS_MGR.DBMS_ERRORCODE" ;
```

**Example:**  
Setting Connection Attributes for an MS SQL Server

```
Compute rc/i8;
rc=sys_mgr.engine("SQLMSS","set connection_attributes mssxyz/ibiusr1,foo"
);
Type "RC from set is <<rc  DBMS Err=<<SYS_MGR.DBMS_ERRORCODE";
```

**Example:**  
Inserting a Row Into a Table (MS SQL)

```
Compute rc/i8;
Type "Inserting row into table MNTTAB2 ";
rc=sys_mgr.engine("SQLMSS","insert into mnttab2 values('X2','XDAT2222');");
Type"ReturnCode=<<rc DBMS Err=<<SYS_MGR.DBMS_ERRORCODE";
```
You will need to test the return code to determine whether the record was inserted successfully (RC = 0).

If you are using MS SQL, and the value you wanted to insert was a duplicate record, you would expect to see the following return codes:

Return Code = -1 DBMS Err = 2627

**SYS_MGR.FOCSET**

Using SYS_MGR.FOCSET, you can set certain environment settings for the WebFOCUS Server. Issue this command from a Maintain procedure to set the desired environment variable, then use a local call from the same procedure to use that setting for your Maintain operations. See *Customizing Your Environment* in the *Developing Reporting Applications* manual for a complete description of the environment settings.

**Syntax:**

**How to Use the SYS_MGR.FOCSET Command**

The syntax is

```plaintext
SYS_MGR.FOCSET("parm","value")
```

where:

- **parm**
  - Is one of the following supported SET commands:
    - CDN
    - COMMIT
    - DATEDISPLAY
    - DEFCENT (DFC)
    - EMGSRV
    - LANGUAGE
    - MESSAGE
    - NODATA
    - TRACEON
    - TRACEOFF
    - TRACEUSER
    - WARNING
    - YRTHRESH
    - PASS
    - USER

    In addition, the parameter `maintain_warning` is included in this command set in order to allow Maintain warning messages to be suppressed.

- **value**
  - Is an appropriate setting for that command.
Example: Setting DEFCENT From a Maintain Procedure

The following code

```
MAINTAIN
COMPUTE MYDATE/YYMD;
SYS_MGR.FOCSET("DEFCENT", "21");
COMPUTE DATE1/YMD='90/01/01';
COMPUTE MYDATE=DATE1;
TYPE "After setting DEFCENT=21, MYDATE=<MYDATE";
END
```

produces the following output:

```
After setting DEFCENT=21, MYDATE=2190/01/01
```

Example: Setting PASS From a Maintain Procedure

The following code will set the password to DBAUSER2:

```
SYS_MGR.FOCSET( 'PASS', 'DBAUSER2.' );
```

Note: When setting a password for DBA access, keep in mind that the last value set from within the application will be in effect for all transactions for that end user session.

Example: Setting maintain_warning From a Maintain Procedure

The following code allows you to display your own error message:

```
case test1.
Compute date1/a10;
Compute date2/mdyy
Sys_mgr.focset("maintain_warning", "off")
compute date2 = date1;
if date2 = '' then
    compute msg = 'Date is not valid';
endcase
```

To set the Maintain warning messages to on, issue the command:

```
sys_mgr.focset("maintain_warning", "on")
```
SYS_MGR.GET_INPUTPARAMS_COUNT

Used in conjunction with the MNTCON EX|RUN –v syntax, the SYS_MGR.GET_INPUTPARAMS_COUNT function retrieves the number of positional parameters passed when invoking a Maintain procedure. If the function is not successful, FOCERROR is set to -1. For information on retrieving the value of a positional parameter, see the SYS_MGR.GET_POSITIONPARM function. See the MNTCON EX and MNTCON RUN commands for information on using the –v option to pass parameters.

Syntax: How to Use the SYS_MGR.GET_INPUTPARAMS_COUNT Command

The syntax for the SYS_MGR.GET_INPUTPARAMS_COUNT command is

\[\text{Var/In} = \text{SYS\_MGR\_GET\_INPUTPARAMS\_COUNT}();\]

where:

\[\text{Var/In}\]

Is the name of the variable with an integer format that you are assigning to the output of the function.

Example: Retrieving the Number of Positional Parameters Passed to a Maintain Procedure

MNTCON EX START1 –v abc, '24 Houston Center'

Target Maintain procedure START1 could include:

\[\text{Posvar/i2}=\text{sys\_mgr\_Get\_InputParams\_count}();\]

Here it returns 2 for computed field Posvar, for positional parameters abc and 24 Houston Center.

SYS_MGR.GET_NAMEPARM

Used in conjunction with the MNTCON EX|RUN –v syntax, the SYS_MGR.GET_NAMEPARM function returns the value of a keyword parameter passed at the time the Maintain procedure was invoked. If the function is not successful, FOCERROR is set to -1. See also the MNTCON EX and MNTCON RUN commands using the –v option to pass parameters.
**Syntax:** How to Use the SYS_MGR.GET_NAMEPARM Command

The syntax for the SYS_MGR.GET_NAMEPARM command is

\[
\text{MyParm/format} = \text{SYS_MGR.GET_NAMEPARM('ParmName')}
\]

where:

- **MyParm/format** is the name of the variable or format that you are assigning to the output of the function.
- **ParmName** is the actual keyword parameter name used when passing the value.

**Note:** The SYS_MGR.GETNAME_PARM function is case-sensitive. Use the same case for the parameter name when retrieving the value as used when passing it.

**Example:** Retrieving the Value for a Keyword Parameter Passed to a Maintain Procedure

MNTCON EX START2 -v ADDR='Cape Canaveral', COUNTRY=USA

Target Maintain procedure START2 could include:

\[
\text{Address/a0=sys_mgr.Get_NameParm('ADDR');}
\]

Here it returns Cape Canaveral to the variable Address.

**SYS_MGR.GET_POSITIONPARM**

Used in conjunction with the MNTCON EX|RUN –v syntax, the SYS_MGR.GET_POSITIONPARM function retrieves the value of positional parameters passed when invoking a Maintain procedure. If the function is not successful, FOCERROR is set to -1. See also the SYS_MGR.GET_INPUTPARAMS_COUNT function to retrieve the number of a positional parameter, and the MNTCON EX and MNTCON RUN commands using the –v option to pass parameters.

**Syntax:** How to Use the SYS_MGR.GET_POSITIONPARM Command

The syntax for the SYS_MGR.GET_POSITIONPARM command is

\[
\text{Var/An} = \text{SYS_MGR.GET_POSITIONPARM(i)};
\]

where:

- **Var/An** is the name of the variable, declared with an alphanumeric format (for example, A0) that you are assigning to the output of the function.
i

Is the position number of the variable whose value you wish to retrieve.

**Example:** Retrieving the Value of a Positional Parameter Passed to a Maintain Procedure

MNTCON EX START3 -v abc, '24 Houston Center'

Target Maintain procedure START3 could include:

```
Parm1/a0=sys_mgr.Get_PositionParm(2);
```

Here it returns 24 Houston Center, the value of the second keyword parameter, for computed field Parm1.

**SYS_MGR.PRE_MATCH**

By default, Maintain first ensures a database row exists before it updates or deletes it and ensures a database row does NOT exist before including a new row. For example, when Maintain processes an INCLUDE, it first issues:

```
SQL SELECT keyfld FROM tablename WHERE keyfld = keyvalue;
```

Then, it only proceeds with the SQL INSERT if the SELECT returned no rows. Many applications are structured so that the designer knows that the row does not exist, so the preliminary SELECT is not needed.

The same is true for DELETE and UPDATE. Only the SELECT must return a row before MAINTAIN continues with the SQL DELETE or SQL UPDATE.

When the application warrants it, you can turn off the preliminary SELECT against relational databases by changing the value of SYS_MGR.PRE_MATCH. For high volume transactions, this can positively affect performance.

**Note:**

- Since you are not checking to see if the row exists or not, it is important to write code that catches errors by inspecting FOCERROR.
- The PRE_MATCH setting is local to the current MAINTAIN procedure. Changing it does not change the value in the parent procedure or in subsequently called procedures.

**Syntax:** How to Set PRE_MATCH

The syntax is

```
SYS_MGR.SET_PRE_MATCH{0|1};
```
or

SYS_MGR.PRE_MATCH = {0|1}

where:

0
   Disables prematching.

1
   Turns on prematching.

To check the current setting for pre-match, use:

SYS_MGR.GET_PRE_MATCH();

or

SYS_MGR.PRE_MATCH;

Example:  Setting PREMATCH Off

Suppose you have a Maintain procedure with the following code:

SYS_MGR.PRE_MATCH = 0;  -* stop pre-selecting
FOR ALL INCLUDE PRODUCTS FROM PRODSTACK;
SYS_MGR.PRE_MATCH = 1;  -* restore

If PRODSTACK has 5000 rows, setting PRE_MATCH to 0 before the INCLUDE reduces the number of database engine interactions from 10,000 to 5,000.

TYPE

The TYPE command writes messages to a file, a web browser, or the Output window of the Maintain Development Environment. The TYPE command is helpful for application debugging, such as tracing application flow-of-control, and for recording an accounting trail. If you wish to display messages to application users, it is recommended that you use forms, which provide superior display capabilities and better control than the TYPE command.

Syntax:  How to Use the TYPE Command

The syntax of the TYPE command is

TYPE [ON ddname] "message" [[|] "message"] ... [;]
where:

**ON** *ddname*

Specifies the logical name of the file that the TYPE message is written to when ON is specified. You must define the ddname (using a DYNAM or FILEDEF command) prior to the first usage. The message string can be up to 256 characters in length. The output starts in column 1. In order to append to an existing file or to write to a file from more than one procedure, append to the file by specifying the appropriate option in the DYNAM command.

If ON *ddname* is omitted, in a:

- Web-deployed application, the message is written to the webpage currently displayed in the web browser.
  - If the message is written from a procedure that resides on a different server than the webpage, the message is prefixed with "(FOC03764) From Server ==> " to indicate that it was posted by a remote procedure.
- Windows-deployed application run from the Maintain Development Environment, the message is written to the Run tab of the Output window.

In addition, if ON *ddname* is omitted and this procedure was called remotely (that is, called using a CALL *procname* AT command), the message will also be copied to the FocMsg stack of the calling procedure.

**message**

Is the information to be displayed or written. The message must be enclosed in double quotation marks ("."). The message can contain:

- Any literal text.
- Variables.
- Horizontal spacing information.
- Vertical spacing information.

The layout of the message is exactly what is specified.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 24.
**Reference:** Commands Related to TYPE

**SAY** writes messages to a file or to the server console. Messages can include multiple expressions of all types.

**Including Variables in a Message**

You can embed variables in a message by prefixing the variable with a left caret (<). Unless the field name is the last item in the string, it must be followed by a space. Maintain does not include the caret and space in the display. For example:

```
TYPE "Accepted:  <Indata(Cnt).Fullname";
```

**Embedding Horizontal Spacing Information**

TYPE information can be placed in a specific column or can be moved a number of columns away from the current position. The following example

```
TYPE "<20 This starts 20 spaces over"
TYPE "Skip <+8 8 spaces within text"
TYPE "Back up <-4 4 spaces and overwrite"
```

results in:

```
This starts 20 spaces over
Skip 8 spaces within text
Back 4 spaces and overwrite
```

**Embedding Vertical Spacing Information**

Lines can be skipped by supplying a left caret (<), slash (/) and the number of lines to be advanced. If the line advance specification is at the beginning of the line, the specified number of lines are advanced before the following text.

```
TYPE "</3 Displays 3 blank lines" |
" before this line"
```

If </number is encountered in the middle of the line, the line feed occurs when </number is encountered.

```
TYPE "This will </2 leave one" |
" blank line before the word leave"
```
Coding Multi-Line Message Strings

Sometimes, a message string needs to be coded on more than one line of a TYPE command. This can occur if indented TYPE lines, spacing information, or field prefixes extend the message string beyond the end of the line. You can wrap a message string onto the next line of a TYPE command if you:

1. End the first line with an ending quotation mark, followed by a vertical bar ( | ).
2. Begin the second line with a quotation mark. For example:

   ```
   TYPE "Name: <Employee(Cnt).First_Name" |
   "<Employee(Cnt).Last_Name" |
   "Salary: <Employee(Cnt).Salary";
   ```

Justifying Variables and Truncating Spaces

To either truncate or display trailing spaces within a field, a left caret (<) or a double left caret (<<) may be used respectively. For character fields, the field values are always left justified. For example:

   ```
   TYPE "*** <Car.Country ***";
   TYPE "*** <<Car.Country   ***";
   ```

produces:

   ```
   *** ENGLAND***
   *** ENGLAND   ***
   ```

For numeric fields, the left caret causes the field values to be left justified, and trailing spaces are truncated. The double left caret causes the field values to be right justified and leading spaces are displayed.

For example

   ```
   TYPE "*** <Car.Seats ***"
   TYPE "*** <<Car.Seats ***"
   ```

produces:

   ```
   *** 4***
   ***   4***
   ```
Writing Information to a File

You can use TYPE commands to write information to a file. The following example writes every transaction record to a log file:

```plaintext
FOR ALL NEXT Emp_ID Last_Name First_Name INTO Stackemp;
COMPUTE Cnt=Cnt+1;
TYPE ON TransLog "<Stackemp(Cnt).Emp_ID " |
"<Stackemp(Cnt).Last_Name" |
"<Stackemp(Cnt).First_Name";
```

The next example places a message into an errors log file if the salary in the stack is greater than allowed:

```plaintext
IF Stackemp(Cnt).Curr_Sal GT Allowamt THEN TYPE ON ErrsFile
"Salary for employee <Stackemp.Emp_ID" |
"is greater than is allowed."
```

The last example writes three lines to the file NoEmpl if the employee is not in the data source:

```plaintext
MATCH Emp_ID;
ON NOMATCH TYPE ON NoEmpl "<Emp_ID"
"<Last_Name"
"<First_Name";
```

UPDATE

The UPDATE command writes new values to data source fields using data from a stack or the Current Area. All of the fields must be in the same data source path. The key fields in the stack or Current Area identify which segment instances to update.

The segment containing the first update field is called the anchor. If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack or Current Area. This enables UPDATE to navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the UPDATE Command

The syntax of the UPDATE command is

```plaintext
[FOR {int|ALL}] UPDATE fields [FROM stack[(row)]] [;]
```

where:

FOR

Is used with int or ALL to specify how many rows of the stack to use to update the data source. When FOR is used, a FROM stack must be supplied. If no FOR prefix is used, the UPDATE works the same way that FOR 1 UPDATE works.
**int**

Is an integer constant or variable that indicates the number of rows to use to update the data source.

**ALL**

Specifies that the entire stack is used to update the corresponding records in the data source.

**fields**

Is used to specify which data source fields to update. You must specify every field that you wish to update. You cannot update key fields. All fields must be in the same path.

**FROM**

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

**stack**

Is the name of the stack whose data is used to update the data source. Only one stack can be specified.

**row**

Is a subscript that specifies the first stack row to use to update the data source.

**;**

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the semicolon, see *Terminating Command Syntax* on page 24.

**Example:** **Using UPDATE**

The UPDATE command can be executed after a MATCH command finds a matching record. For example:

```
MATCH Emp_ID;
ON MATCH UPDATE Department Curr_Sal Curr_Jobcode Ed_Hrs FROM Chgemp;
```
Consider an application used when an employee changes his or her last name. The application user is prompted for the employee ID and new last name in a form. The user enters the name and triggers the ChngName function. If the employee is in the data source, ChngName updates the data source. If the employee is not in the data source, ChngName displays a message asking the user to try again.

```
CASE ChngName
    REPOSITION Emp_ID;
    MATCH Emp_ID;
    ON MATCH BEGIN
        UPDATE Last_Name;
        COMMIT;
        Winform Close;
    ENDBEGIN
    ON NOMATCH BEGIN
        TYPE "Employee ID <Emp_ID was not found"
        "Try again";
    ENDBEGIN
ENDCASE
```

The command can also be issued without a preceding MATCH. In this situation the key field values are taken from the FROM stack or the Current Area and a MATCH is issued internally. When a set of rows is changed without first finding out if the set already exists in the data source, it is possible that some of the rows in the stack will be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. For all rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work, and a ROLLBACK issued if the entire set is not accepted.

**Reference:** Usage Notes for UPDATE

- Key fields cannot be updated.
- There can only be one input or FROM stack in an UPDATE command.
- When an UPDATE command is complete, the variable FocError is set. If the UPDATE is successful, FocError is set to zero. If the records do not exist, and are therefore unchanged, FocError is set to a non-zero value and (if the UPDATE is set-based) FocErrorRow is set to the number of the row that failed.
- Maintain requires that the data sources to which it writes have unique keys.

**Reference:** Commands Related to UPDATE

- **COMMIT** makes all data source changes since the last COMMIT permanent.
- **ROLLBACK** cancels all data sources changes made since the last COMMIT.

### Update and Transaction Variables

After the UPDATE is processed, the internal variable FocError is given a value. If the UPDATE is successful, FocError is zero. If the UPDATE fails (that is, the key values did not exist in the data source) FocError is set to a non-zero value, and (if the UPDATE was set-based) FocErrorRow is set to the number of the row that failed. If at COMMIT time there is a concurrency conflict, FocError and the internal variable FocCurrent are set to non-zero values.

### Example: Using Stacks

In the following example, the user enters many employee IDs and new names at one time. Rather than performing a MATCH on each row in the stack, this function checks FocError after the UPDATE command. If FocError is zero, a COMMIT is issued and the function is exited. If FocError is not zero, another function, which tries to clean up the data, is performed. The IF command, which starts at the beginning of the function, checks to see whether there are any rows in the stack. If the stack does contain have any rows, a form is displayed allowing the user to enter new data. If the stack contains rows, the user has made a mistake, so a different form is displayed allowing the user to edit the entered data.

The Maintain procedure contains:

```c
STACK CLEAR Namechng;
PERFORM Chngname;
CASE Chngname
  IF Namechng.FocCount LE 0
    THEN Winform Show Myform1;
    ELSE Winform Show Myform2;
  FOR ALL UPDATE Last_Name FROM Namechng;
  IF FocError EQ 0 BEGIN
    COMMIT;
    GOTO ENDCASE;
  ENDBEGIN
  PERFORM Fixup;
  GOTO Chngname;
ENDCASE
```

### Data Source Position

A Maintain procedure always has a position either in a segment or before the beginning of the chain. If positioned within a segment, the position is the last record successfully retrieved on that segment. If a retrieval operation fails, then the position of the data source remains unchanged.
If an UPDATE is successful, the data source position is changed to the last record it updated. If an UPDATE fails, the position is at the end of the chain because the MATCH prior to the UPDATE also fails.

Unique Segments

The UPDATE command treats fields in unique segments the same as fields in other types of segments.

Winform

The Winform command controls the forms that appear on the screen. Forms are used to edit and display data. They act as a user interface, whereas a procedure controls the application logic and use of data.

Syntax: How to Use the Winform Command

The Winform command performs three tasks:

- Displaying and controlling forms.
- Setting form and form control properties.
- Querying form and form control properties.

The syntax of the Winform command for displaying and controlling forms is

```
Winform command formname [;]
```

where `command` is one of the following:

Show

Makes the specified form active. It displays the form and transfers control to it, enabling an application user to manipulate the controls (of the form), such as buttons and fields. In a Windows-deployed application, if other forms are currently displayed on the screen, the specified form is displayed on top.

Show_Active

Can be used for clarity. It is functionally identical to Show.

Show_Inactive

In Windows-deployed applications, this displays the specified form without making it active. Because the form is inactive, control passes to the following command, not to the form. You can use this to change the initial properties of a form, and its controls, dynamically at run time before the form is displayed.
In web-deployed applications, you can use this to change the initial properties of a form, and its controls, dynamically at run time. SHOW_INACTIVE does not display the form.

In web-deployed applications, this option does not apply.

**Reset**

Resets a form and its controls to their original properties. All selectable controls, such as list boxes, check boxes, and radio buttons, return to their default selections.

**Refresh**

Repopulates the data values of the form as if control had returned to the form from an event handler, but without making the form active.

In web-deployed applications, this option does not apply.

**Close_All**

Closes all forms. The form environment remains active.

**Close**

Closes the chain of forms from the currently active form back up to the specified form. If you do not specify a form, the command closes only the currently active form.

The close operation does the following:

- Passes control directly to the beginning of the chain, to the point just following the Winform Show command that called the specified form.

- Removes closed forms from the screen.

**Show_And_Exit**

Displays the specified form and then immediately terminates the application. This enables you to end an application while displaying a final form that remains on the screen. In an application deployed in a web browser, this avoids returning the end user to the launch form. Any client-level logic (such as hyper text links, JavaScript and VBScript functions, and Java applets) will remain active, but all native Maintain logic such as event handlers will not respond because the application has terminated. SHOW_AND_EXIT is not applicable to Windows-deployed applications.

The syntax of the Winform command for changing a form control property is:

```
Winform Set formname[.controlname].property TO value [;]
```

The syntax of the Winform command for querying a form control property is

```
Winform Get formname[.controlname].property INTO variable [;]
```
where:

formname
Is the name of the form.

controlname
Is the name of the form control whose property you wish to set or get. Omit the control name if you are changing a property for an entire form, otherwise you must specify it.

property
Is any valid property.

value
Is a value that is valid for the specified property.

variable
Is any scalar variable (a user-defined field or a stack cell) to which you will assign the value of the specified property of the specified form or control.

;
Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the semicolon, see Terminating Command Syntax on page 24.

For information about form and control properties, see Form and Control Properties Reference in the Developing WebFOCUS Maintain Applications manual. For information about using the WINFORM SET and GET commands, see Dynamically Changing Form Control Properties on page 200.

Reference: Commands Related to Winform

- NEXT retrieves sets of data from a data source into a stack. You can then display the data in a form.

- TYPE displays messages on the screen or writes them to a file.
Displaying Default Values in a Form

If a form displays a variable that has not been assigned a value, the form will display the default value. The default value of the variable is determined by its data type and whether it was defined with the MISSING attribute:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default value without the MISSING attribute</th>
<th>Default value with the MISSING attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character/Alphanumeric</td>
<td>space</td>
<td>null</td>
</tr>
<tr>
<td>Numeric</td>
<td>zero</td>
<td>null</td>
</tr>
<tr>
<td>Date and time</td>
<td>space</td>
<td>null</td>
</tr>
</tbody>
</table>

A null value is displayed as a period (.) by default. You can specify a different character using the SET NODATA command.

Dynamically Changing Form Control Properties

You can change many properties of forms and controls at run time using the Winform Set command, and can determine the current state of those properties using the Winform Get command. You can set other properties using functions.

If you want to change the properties of a form at run time before the form is displayed, you can first issue the Winform Show_Inactive command, then issue commands to set form and control properties, and finally issue a Winform Show command. In Windows-deployed applications, Winform Show will make the form active. In web-deployed applications, it will display the form. If you wish to change the properties of a form in response to user activity in the form, you can trigger a function containing Winform Set commands and function calls from those user events. You cannot dynamically set the properties of a form before it has been opened with either a Winform Show or Winform Show_Inactive.

For example, you could develop a data entry function that determines whether a user has entered data into a field. If the user has not, you could use the Winform Set command to change the color of the field and give it focus, effectively drawing attention to it and making it the target of any keyboard activity.

For information about form and control properties, see the Developing WebFOCUS Maintain Applications manual.
The following is a list of error messages that you may encounter when using WebFOCUS Maintain.

**In this appendix:**

- Error Messages

### Error Messages

(FOC03601) ERROR AT OR NEAR LINE%1 IN PROCEDURE %2.

An error occurred near the specified line number. The subsequent error message(s) should indicate the problem.

(FOC03602) Warning. Ignoring redefinition of %1 field %2.

In MAINTAIN, once a field has been established with a format, the format cannot be changed. Most likely, this MAINTAIN has a field name that has been given two different formats. Either change one of the field names or omit the format.

(FOC03603) Types %1 and %2 differ; built in conversion supplied.

(FOC03604) %1 is not a valid format.

The format specified after a field name is not valid. FOCUS supports A1 through 256, I1 through 9, D1 through 15, P1 through 15, F1 through 9 and date formats. Formats may also contain any of the allowed edit options.

(FOC03605) %1 is not recognized.

MAINTAIN has encountered an object, such as a field name, stack name, or case name, that does not exist. Check for spelling errors or any other errors that might have occurred.

(FOC03606) Incompatible types %1 and %2.

(FOC03607) %1 is an invalid type for operator %2, expecting %3 type.

(FOC03608) Invalid constant '%1'.

The value specified could not be interpreted by MAINTAIN as a constant of any FOCUS type. Check the specification of the constant for any typographical errors.
(FOC03609) Wrong argument count in call to case %1. Got %2, expected %3.

The definition of a MAINTAIN case specifies the number and types of parameters it expects. The invocation of the case that was flagged as an error did not comply with the definition of the case as regards to the number and/or types of the parameters passed.

(FOC03610) %1 is not a field in (or member of) type %2.

The most common example of this error is referencing a field in a stack when the stack does not contain that field as a column.

(FOC03611) %1 is not a method in type %2.

(FOC03612) Type %1 is not a stack or collection type.

(FOC03613) EVENT not allowed outside of method context or in STARTUP.

(FOC03614) %1 is not an event in type %2.

(FOC03615) Type %1 is not defined.

(FOC03616) Warning. Too many values, ignoring value#%1.

There are too many values in a nested initialization for the data type. Extra values are ignored.

(FOC03617) Too many subscripts on type %1.

(FOC03618) Unable to analyze type reference.

(FOC03619) MASTER file has changed since procedure :%1: was compiled.

One of the MASTER files mentioned in the FILE statement has changed since the MAINTAIN was compiled. The Maintain procedure mentioned in the error message needs to be compiled again.
(FOC03620) Pointer variables must be initialized.
(FOC03621) Bad pointer type reference %1.
(FOC03622) Bad collection type reference '%1 OF %2'.
(FOC03623) 'RETURNS ...' not allowed on case %1 in type %2, ignoring.
(FOC03624) Duplicate definition of case %1 in type %2, ignoring.
(FOC03625) Duplicate definition of event %1 in type %2, ignoring.
(FOC03626) Duplicate definition of type %1, ignoring.
(FOC03627) Duplicate definition of case %1, ignoring.
(FOC03628) Type name %1 is not allowed because it is a FOCUS format.
(FOC03629) Both initialization and STARTUP for global %1.
(FOC03630) Compiler Error: %1.
(FOC03631) (Internal Error) No Info Node.
(FOC03632) Internal Error.
(FOC03633) Stack : %1 : contains multiple paths.

In MAINTAIN, a single stack is restricted to containing data from only one path. Examine the database statements that reference the stack in question to determine where the stack received data from different paths.

(FOC03634) Database Statement does not reference the Database.

None of the fields in the database statements referred to fields in the database. Check the spelling of the field in the statement. Alternatively, if the field name is ambiguous, qualify it with the segment, Master File name, or both.

(FOC03635) %1 : Is not a Database Entity.

The field in question appears in a statement that requires a database field. The field specified could not be interpreted as a database field.

(FOC03636) %1 : Is Unsupported Syntax.

(FOC03637) A Database Statement may only reference one Path.

In MAINTAIN, only one path may be referenced by any single database statement. If data is required from multiple paths, process each path with separate statements. Remember that a single stack has the same single path only restriction.
(FOC03638) $1 area overflow.
(FOC03639) Invalid Combination of Copy Arguments.

If a Copy specified Current as the From or Into, then the statement may not have a For or Where clause.

(FOC03640) On Match/Next Include is invalid syntax.

In MAINTAIN, you must avoid multiple instances of the same key in the database. As a result, syntax like ON MATCH INCLUDE is problematic and flagged as an error.

(FOC03641) On NoMatch/NoNext Update/Delete is invalid syntax.

Database statements, such as UPDATE and INCLUDE, operate on a record in the database. Statements like ON NOMATCH UPDATE have no meaning.

(FOC03642) Root of retrieval may not be a Unique segment.
(FOC03643) There are no databases in scope for the statement.
(FOC03644) Field : $1 : is not updateable.

Certain fields in a database are marked as non-updateable. Key fields in FOCUS databases are such fields.

(FOC03645) WHERE clause syntax : $1.
(FOC03646) FOR (update/delete/include/revise) requires a FROM stack.

The FOR UPDATE statement is designed to operate on a set of records. Such a set must be specified (through the FROM clause) for the statement to make sense.
(FOC03647) %1 type information incomplete for %2.
(FOC03648) FOR retrieval requires an INTO stack.
(FOC03649) Undefined Stack : %1 : in %2 clause.
(FOC03650) Syntax Error at or near line %1 in %2.
(FOC03651) The command %1 is not yet implemented.
(FOC03652) (Internal Error) in Maintain Parser.
(FOC03653) Cannot Display Non-Scalar Object.
(FOC03654) Undefined CASE name: %1.
(FOC03655) Invalid WINFORM command: %1.
(FOC03657) Subscript must be an integer for type %1.
(FOC03658) Invalid type/format for argument #%1 incall to CASE %2.
(FOC03659) STACK CLEAR error: %1.
(FOC03660) WINFORM Error on %1: SHOW command must precede HIDE.
(FOC03661) WINFORM Error on %1: SHOW command must precede UNHIDE.
(FOC03662) (Internal Error) Unable to read UWS object at address %1.
(FOC03663) (Internal Error) Unable to read '%1' for function '%2'.
(FOC03664) Error processing arguments to external function '%1'.
(FOC03665) Error loading external function '%1'.
(FOC03666) : %1 : Is an Invalid Type Column Position.

Subsequent data to be typed is ignored.

(FOC03667) Type Statement Area Overflow near column : %1 :.

There is a limit of 256 bytes on the size of a single line of formatted output in a TYPE statement. Try inserting "</n" to break the large line into smaller pieces.
(FOC03668) Winform Error: Source data, %1, is not in %2 format.
(FOC03669) Winform Error: Received data, %1, is not in %2 format.
(FOC03670) Target of CONTAINS/OMITS, %1, is longer than source, %2.
(FOC03671) DEPENDENTS ignored on CASE %1.
(FOC03672) Invalid Fieldname following FOC.
(FOC03673) : %1 : Exceeds maximum length for a virtual fieldname.
(FOC03674) ?flat may only operate on real fields.
(FOC03675) Error parsing file : %1 :
(FOC03676) : %1 : Is not a valid compile time function.
(FOC03677) A COPY Statement with a WHERE clause requires a FOR.
(FOC03678) DB Format Error: Data from Database, %1, is not in %2 format.
(FOC03679) DB Format Error: Data for Database, %1, is not in %2 format.
(FOC03680) %1 is not an acceptable value for field %2.
(FOC03681) Syntax Error near column %1.
(FOC03682) Error In FIND. : %1 : does not specify a Database Field.
(FOC03683) Invalid Value : %1 does not lie between %2 and %3.
(FOC03684) %1 is Bad ProcName: Exceeds Maximum Length.

The procedure name in an EXEC Statement or a METHOD in the Master File exceeds the maximum supported name length, currently equal to 31.

(FOC03685) Unable to Create Execution Manager.

The STARTUP Method failed for the Execution Manager Object, probably due to a resource outage. The Execution Manager is created to process the EXEC statement, or a Method in the Master File, or to send and receive parameters.
(FOC03686) Compilation failed for called Procedure: %1.
(FOC03687) Load failed for compiled image of called Proc: %1.
(FOC03688) Run failed for called Procedure: %1.
(FOC03689) Called Proc, %1, wants %2 input parms but received %3.
(FOC03690) Called Proc, %1, wants %2 Output parms but was called w/ %3.
(FOC03691) %1 Parm number %2 cannot be mapped from caller to callee.
(FOC03692) Run-time Error : %1.
(FOC03693) WARNING : Divide by zero!
(FOC03694) Stack index is less than or equal to zero.
(FOC03695) Parameter Type Not Supported For Procedure Calls.
(FOC03696) Parameter Type Map Exceeds Maximum Length.
(FOC03697) Fully Qualified Name Required For Type Map Too Long.
(FOC03698) WARNING!... %1 Parm number %2: Partial Type Match.

The stack composition of the CALLER stack is not identical to that of the stack called.

(FOC03699) Fatal Error...INTO stack requires HTML column name.

For the EXEC statement receiving PCHOLD FORMAT HTML/HTMTABLE records from the CALLEE, a column name of HTML and format A250 is required.

(FOC03699) Fatal Error...%1 Stack Number %2 : No Columns Match.

The CALLER stack contains no columns equivalent to those columns in the called stack.

(FOC03700) FATAL ERROR!... %1 Parameter %2 : Incompatible Types.
(FOC03701) Invalid SET %1 value for MAINTAIN.

Maintain requires the FOCUS field name setting to be NEW or NOTR. A setting of OLD will produce this message. Similarly, Maintain requires the QUALCH to be a period (\'\).'.

(FOC03702) Warning. No position on parents of segment: %1 ::

An attempt was made to either MATCH or NEXT (explicitly or implicitly) on the segment specified without a position on the parent segments. As a result no records could be retrieved.
Warning. Cannot find the IMPORT file : %1:.

Cannot find the file : %1 :.

Sink Name Conflict on Database : %1 :.

In a Maintain application, only one sink machine may be specified for any one FOCUS database. Maintain detected the same database being specified with different ON sink names. The application is terminated.

WINFORM Error on %1: CLOSE not allowed for suspended form.

WINFORM Error on %1: SHOW command must precede CLOSE.

WINFORM Error on %1: Duplicate SHOW command for a form.

WINFORM Error on %1: INACTIVE not allowed for active form.

WINFORM Error on %1: SHOW command must precede REFRESH.

WINFORM Error on %1: HIDE is not allowed for active form.

WINFORM Error on %1: UNHIDE is meaningless for displayed form.

EDIT CONTROL Error on %1: Associated variable is not an FBIT.

GRID Error on %1: Associated variable is not a Stack.

STATIC CONTROL Error on %1: Associated variable is not an FBIT.

LISTBOX Error on %1: Associated variable is not a Stack.

RADIOGROUP Error on %1: Associated variable is not a Stack.

Error on %1: Object is not a Rectangle.

Error on %1: Object is not a Window.

Variable :%1: is not a stack. It cannot be sorted.

:%1: is not a column in stack :%2:. Cannot sort on it.

Command :%1: failed. Out of Memory.

:%1: is not a database field.

The update command operates only on fields.

Invalid Stack Copy Argument.

The Copy command may only be used for moving structured types. Either the From or the Into argument is a scalar.
(FOC03725) :%1: is explicitly declared and cannot be inferred.
(FOC03726) DBA Error. User does not have %1 access rights for :%2:.
(FOC03727) Datatypes not supported for computational/comparison operation.
(FOC03728) Segment :%1: is not writable.
(FOC03729) Error processing return value for external function ':%1'
(FOC03730) :%1: is/are not supported in Maintain.
(FOC03731) Error Allocating Memory for X-Maintain Execution.

    This error may occur during the compilation or execution phase.

(FOC03732) Error Binding to Child Procedure: %1.

    The bind failure is only fatal during execution phase.

(FOC03733) Communications Error in X-Maintain Processing.

    Using the appropriate communications trace may yield more detail.

(FOC03734) Error occurred on MAINTAIN srvr during remote processing.
(FOC03735) Error Occurred Building Outbound X-Maintain Data Stream.
(FOC03736) Server returned inconsistent Bind Reply Data Stream.
(FOC03737) Server returned unrecognized Data Stream.
(FOC03738) Comm Failure Attempting Send to Proc %1 at Srvr %2.
(FOC03739) Failure Attempting Communications Initialization.
(FOC03740) Connect to Proc %1 at Srvr %2 failed, no free agent.

    Connection to remote server failed.
(FOC03741) Comm Fail Receiving Message from Proc %1 at Srvr %2.
(FOC03742) Communications Driver returned unrecognized Event.
(FOC03743) Error Exchanging Maps With Proc %1 at Srvr%2.
(FOC03744) Error Packing Data For Shipment to Proc %1 at Srvr %2.
(FOC03745) Error Receiving Data From Proc %1 at Srvr %2.
(FOC03746) Invalid Execute Reply message from Proc %1 at Srvr %2.
(FOC03747) Error Message %1 issued for Proc %2 at Srvr %3.
(FOC03748) Too Many Databases Specified.

Currently, a Maintain procedure may specify up to and including 16 databases on the command line. In this case, more than 16 were specified.

(FOC03749) %1 FOCCOMP is 'old'. To regain start up speed use RECOMPILE.

In this release, the format of the MAINTAIN FOCCOMP has changed. You can still use RUN to execute a MAINTAIN FOCCOMP but the time to first screen will be slow (similar to EX of MAINTAIN FOCEXEC). To regain the start up speed, use RECOMPILE on the old FOCCOMP to produce a new FOCCOMP.

(FOC03750) COMBOBOX Error on %1: Associated variable is not a Stack.
(FOC03751) PICTUREBOX Error on %1: Associated variable is not an FBIT.
(FOC03752) PICTUREBUTTON Error on %1: Associated variable is not an FBIT.
(FOC03753) Warning: Missing Closing Quote near Column %1.
(FOC03754) Duplicate item name %1 in type %2.

Items names must be unique.

(FOC03755) Last argument to %1 must be a variable.

The last argument in a user written subroutine call must be a variable.

(FOC03756) Cross-reference to file %1 on different SU.

All of the cross-references of a file must be on the same sink machine. Check the USE list.

(FOC03757) Acceptable range of values is between %1 and %2.
(FOC03758) Communications Software Not Installed Or Not In Path.

The iWay API is not available to Parent Maintain attempting cross machine CALL.
(FOC03759) An Invalid Stack was provided to the GetAccept Method.

The stack passed by an application to the GetAccept method on a field must have an ALPHANUMERIC field of sufficient width as its first column.

(FOC03760) SetFocus can't be applied before Winform %1 is shown.

(FOC03761) Fatal Error: %1 %2.

The above error should never occur. If it does, write down the exact message description and report the problem to the Information Builders office in New York.

(FOC03762) Can't use FIND with non-indexed field %1.

The field specified must be an indexed field.

(FOC03763) Prior Message Produced by Proc %1 on Svr %2.

(FOC03764) From Server ==> %1.

(FOC03765) Error Loading Compiled Procedure: %1.

The COMPILED image of the specified procedure cannot be loaded.

(FOC03766) Client/Server mismatch for file %1.

The number of fields for this file does not match. Possibly due to different Master Files on the client and server or a DBA field restriction.

(FOC03767) FIND error. Fields %1 and %2 are incompatible.

The "from" and "to" fields have different lengths.

(FOC03768) Error writing to %1 FOCCOMP.

(FOC03769) Unable to open %1 FOCCOMP for writing.

(FOC03770) Unable to open %1 FOCCOMP for reading.

(FOC03771) Error reading from %1 FOCCOMP.

(FOC03772) Invalid version for %1 FOCCOMP. Please RECOMPILe.

The FOCCOMP was produced with an older version of MAINTAIN. Use the RECOMPILE command to create an updated FOCCOMP.

(FOC03773) COMPILe failed with return code %1.

(FOC03774) Database position invalidated for file %1.

A child (called) MAINTAIN has performed a database operation that invalidated the position of the parent MAINTAIN.
(FOC03775) Dynamic cross-reference for file %1 not supported.

The closest "real" parent of the dynamic cross-referenced segment is not using the local direct FOCUS access mechanism.

(FOC03776) File-Transfer-Utility cannot obtain enough memory.
(FOC03777) FTP client attempted operation with unknown server.
(FOC03778) FTP Put has failed.

No server acknowledgement has been received.

(FOC03779) FTP utility has encountered an invalid data stream.
(FOC03780) FTP utility: internal error building x-mit packet.
(FOC03781) FTP utility encountered error attempting to read file.
(FOC03782) FTP encountered error transmitting file.
(FOC03783) FTP request for FileList has failed.
(FOC03784) FTP failure writing file.
(FOC03785) FTP failed connecting to server.
(FOC03786) FTP failure: Communications link is not installed.
(FOC03787) FTP failure: Communications failure.
(FOC03788) FTP failure: Server reports an error in processing.
(FOC03789) FTP fail: internal error building data stream.
(FOC03790) FTP fail: Send Fail.
(FOC03791) FTP fail: Communications Initialization Error.
(FOC03792) FTP fail: Error receiving response from server.
(FOC03793) Warning: Null Stack Copy Detected.

The specified COPY command (includes database commands that specify a FROM stack) has not found any columns in the two stacks that match. No data will be moved. If field names and formats match, check the missable attribute.

(FOC03794) Protocol compatibility error with SU ( %1).

The client requires services that are unavailable on the specified server. The SU (sink machine) should be upgraded to the proper revision level.

(FOC03795) Invalid PF-key mapping.
(FOC03796) Master File :%1: has already been specified.

The list of Master Files on the Maintain command line may not have repetitions.
(FOC03797) File %1 is write-protected.

    The user has attempted to UPDATE, INCLUDE, or DELETE to a write-protected file.

(FOC03798) Call Transmission Limit Exceeded.

    There is currently a maximum amount of data (this includes the data describing the data) that can be passed in a CALL statement.

(FOC03799) Open for file :%1: failed.

    The file specified could not be opened. If it was a DDNAME, check that a FILEDEF was issued for it.

(FOC03800) Duplicate Declaration of %1 Variable %2.

    The variable specified has been declared more than one time in the specified scope level.

(FOC03801) Maintain EXEC RPC failure.

(FOC03802) Maintain EXEC Receive failure.

(FOC03803) Maintain EXEC Connect failure.

(FOC03804) Maintain EXEC Bind failure.

(FOC03805) Local Maintain EXEC not supported on this platform.

(FOC03806) FTP failure: compile error.

(FOC03807) FTP failure: error deleting file.

(FOC03808) Warning: Boolean operation on non-boolean expression.

    The expression may not produce the expected results.

(FOC03809) Warning: 'IS NOT' encountered, try 'IS_NOT'.

    The expression may not produce the expected results.

(FOC03810) Error overriding %1 %2::%3 : signatures are not identical.

    The event or method definition overrides a method or event in the base class that has a different parameter list or return value.

(FOC03811) Incomplete class definition '%1'.

    An incomplete or undefined class was encountered while processing the formal parameter list of a method.

(FOC03812) EXPORT modules cannot be executed directly.

    Programs that are specified as MODULE EXPORT cannot be executed directly.
(FOC03813) Invalid type %1 for argument %2 to user written subroutine %3.

User-written subroutines can only have arguments that are FOCUS types. (No user defined types or MAINTAIN extended types allowed.)

(FOC03814) Unclosed multi-line comment.

Multi-line comments that start with '$*' must be closed with '*$'.

(FOC03815) Stack Sort BY field :%1: does not specify a column.

The most likely cause is that the file name or a segment name in the stack is the same as the field name you are trying to sort by. Augmenting the qualification of the field name will fix the problem.

(FOC03816) Disconnect failed.

(FOC03817) Warning: %1 is outside of the century window.

(FOC03818) ENDREPEAT encountered outside of a REPEAT loop.

(FOC03819) Date format error: %1 cannot be greater than %2.

(FOC03820) Warning: Assuming %1 is an abbreviation for %2.

When a field in a structure is not found, MAINTAIN will look for nested fields with the same name as an abbreviation. In certain cases, this may lead to unintended behavior. The number or size of the local variables, or the size of an expression may be too large.

(FOC03821) Local variable space overflow.

The number or size of the local variables, or the size of an expression may be too large.

(FOC03822) Expression too large.

There are too many terms in an expression.

(FOC03823) NULL object handle encountered.

A handle (@ variable) is being referenced that has not been assigned to an object.

(FOC03824) Memory allocation failure for %1.

An internal function was unable to allocate sufficient memory for task completion. Try increasing the run-time memory allocation. If the error condition persists, notify the Information Builders office in New York.

(FOC03825) Recursive CALL of Maintain :%1: detected. Not Supported.

Maintain does not support recursive calls. This includes direct and indirect recursive calls.
(FOC03826) Expression has no value.

MAINTAIN requires a value, but the expression coded does not produce a value.

(FOC03827) Warning: Assuming %1 is a FUSELIB function.

The function being referenced may not be available on all platforms.

(FOC03828) From Server %1 ==> %2.

(FOC03829) An EXEC Statement must specify a Server.

An Exec statement requires an AT clause.

(FOC03830) Error encountered while parsing master file :%1:.

(FOC03831) Unrecognized MNTCON Command.

(FOC03832) Unable to RUNIMAGE :%1:. Image not previously loaded.

For an application to be invoked through RUNIMAGE, there must have been a LOADIMAGE for it previously in the server profile.

(FOC03833) Internal Error detected in MASPAR :%1:.

(FOC03834) Unable to PREPARE Maintain Application Server.

(FOC03835) Unable to START Maintain Application Server.

(FOC03836) LOADIMAGE for :%1: Failed.

Check for other error messages to determine the cause of the failure.

(FOC03837) LOADIMAGE for :%1: was Successful.

(FOC03838) Cannot prepare specified Maintain for Application Server.

The Maintain procedure in question has requirements that conflict with the Application Server architecture. Check subsequent error messages to determine which Maintain procedure is at fault. Most likely, the cause is FOCUS DB access, DBA in the Master File, or DEFINES in the Master File.

(FOC03839) AppServer command :%1: invalid in state :%2:.

Certain application server commands are invalid in certain application server states. For example, a running application server cannot accept a COMPILE command.

(FOC03840) Called Procedure :%1: has not been preloaded, Cannot Invoke.

Once the Maintain Application Server has been started, all Maintain procedures that are invoked must have been preloaded during the preparation phase of the server.
(FOC03841) A Maintain Remote CALL Server must specify PRIVATE deployment.

Reconfigure the server. The most likely cause of confusion is due to the fact that the Maintain Scaleable Application Server requires POOLED deployment. The Maintain Remote CALL Server is a different type of server.

(FOC03842) Packed decimal precision loss during call.

(FOC03843) Client Parameter at Index: %1 Precision: %2 Row: %3, Column: %4.

(FOC03844) Input Parameter Data: %1.

(FOC03845) Does Not Match Server Parameter Precision.

(FOC03846) Server Focexec: %1.

(FOC03847) %1 %2 %3 %4 %5 %6 ].

(FOC03848) Maintain image of files does not match current file list.

A non-Maintain request (for example, TABLE) has caused file descriptions to move in memory since the Maintain procedure was loaded. Use the USE command in your profile to latch the files into relative positions before running any Maintain procedures.

00000(FOC03849) Compiled maintain image not loadable: %1.

00000(FOC03850) Prior traced statement: %1.

When an error or warning occurs, the last statement recognized by the Maintain statement trace facility is displayed. The granularity of the statement tracing can be increased by using the DEBUG option when the procedure is compiled. (The syntax is MNTCON DEBUG COMPILE procnme.)

00000(FOC03851) Database rollback forced due to previous error.

00000(FOC03852) Warning: %1 of fex or winform size-limit already reached.

Over 90% of AST nodes are used already. Adding more complexity to the fex or winform could exceed AST node limit.

00000(FOC03853) Failure opening an IMPORT.

00000(FOC03854) MNTCON command %1 requires a filename argument.

(FOC03855) FATAL ERROR!... %1 Parameter %2: Stack cannot be subscripted.

(FOC03856) FCM file contains no Web Debug information.
## Index

- comment delimiter 24
- subtraction operator 29
* multiplication operator 29
** exponentiation operator 29
$ comment delimiter 24
/ division operator 29
\ character in character expressions 46, 47
\n character in character expressions 46
+ operator 29
| and || characters in character expressions 46
$ comment delimiter 24
$ comment delimiter 24

### A

A0 data type 48, 49
A0 variables 65, 107, 124
passing between procedures 65, 107, 124
addition in date expressions 39
addition operator 29
ALL keyword 75, 79, 84, 87, 118, 142, 163, 171, 193
in COMPUTE command 75
in COPY command 79
in DECLARE command 84
in DELETE command 87
in INCLUDE command 118
in NEXT command 142
in REPEAT command 163
in REVISE command 171
in UPDATE command 193
alphanumeric expressions 45, 46, 47
backslash (\) character 46, 47
concatenating strings 46
escape character 46
evaluating 46

### alphanumeric format
variable length 48, 49
alphanumeric format (MODIFY)
passing A0 variables to procedures 107, 124
AND keyword 124, 142
in MAINTAIN command 124
in NEXT command 142
AND logical operator 50, 51
applications
controlling environment 181
debugging 111
assigning values to variables (Maintain) 58
AT keyword 65, 107
in EXEC command 107
AT keyword in CALL command 65

### B

backslash (\) character in character expressions 47
BEGIN command (Maintain) 63, 64
nested BEGIN blocks 64
Boolean expressions 50

### C

CALL command 64, 65, 125, 126
MAINTAIN command and 125, 126
canceling commands (Maintain) 24
CASE command 69, 70, 71
PERFORM command and 71
case sensitivity 18
cases (Maintain) 69, 73, 79
calling using COMPUTE 79
return values 73
Top function 73
CDN (Continental Decimal Notation) 33, 184
setting in a Maintain procedure 184
character expressions 45, 46, 47
\ character 46, 47
\n character 46
character expressions (continued)  
| and || characters 46  
backslash (\) character 46, 47  
concatenating strings 46  
double quotation marks 46, 47  
escape character 46  
evaluating 46  
single quotation marks 46, 47  
character format 48, 49  
variable length 48, 49  
character strings 46, 47  
classes  
libraries 141  
CLEAR keyword 178  
in STACK CLEAR command 178  
CLOSE keyword in WINFORM command 197  
CLOSE_ALL keyword in WINFORM command 197  
columns in stacks with virtual fields 78  
command types 23  
multi-line 23  
commands 23, 24  
canceling 24  
comment delimiters 24  
dollar sign asterisk 24  
double dollar sign 24  
comments 22, 24  
Maintain procedures 22, 24  
COMMIT command (Maintain)  
setting parameter from a Maintain procedure 184  
COMMIT command (MODIFY) 73  
COMPILE command (Maintain) 130  
COMPUTE command (Maintain) 74, 75, 78, 118  
creating user-defined fields 78  
IF command and 118  
conditional actions (Maintain) 60  
conditional expressions 52, 118  
COMPUTE commands 118  
CONTAINS logical operator 51, 146  
in NEXT WHERE phrase 146  
Continental Decimal Notation (CDN) 33  
COPY command 79  
CURRENT keyword in COPY command 79  
D  
DATA keyword 75, 84  
in COMPUTE command 75  
in DECLARE command 84  
data source stacks 19, 78, 79, 121, 141  
copying rows 79  
libraries and 141  
naming 19  
virtual fields 78  
data sources  
libraries and 141  
navigating 149  
position 169  
updating 170, 193  
data types  
matching in function parameters 71  
date and time expressions 34  
date expressions 34, 35, 36, 37, 38, 39  
addition and subtraction in 39  
components 37  
constants in 37  
evaluating 35  
extracting 37  
formats 34, 36  
manipulating in date format 36  
operand format 38  
date-time data types 40, 42, 43, 44  
assigning 42  
comparing 42  
describing 40  
functions 43  
ISO standards 44  
missing values and 42  
date-time formats 40  
date-time values 40  
assigning 40  
DATEDISPLAY parameter 184  
DBMS_ERRORCODE command 182  
DECLARE command 84
DECRIPT command 104
DEFCENT parameter 184, 185
DEFINE attribute
  in Master File 78
DELETE command 87
  deleting data 87
DFC keyword 75, 84
  in COMPUTE command 75
  in DECLARE command 84
DFC parameter 184
directory paths 46, 47
DIV operator 29, 30
division operator 29
dollar sign asterisk comment delimiter 24
double dollar sign comment delimiter 24
DROP keyword 65, 107
  in CALL command 65
  in EXEC command 107
DROP TABLE command from Maintain 183
duplicate names 19

E
ELSE keyword 52, 115, 117
  in conditional expressions 52
  in IF command 115
EMGSRV parameter 184
ENCRIPT command 104
END command 106
END keyword
  in GOTO command 112
ENDBEGIN keyword 63
ENDCASE 69, 70, 112
  command 69, 70
  keyword in GOTO command 112
ENDREPEAT 112, 163
  command 163
  keyword in GOTO command 112
ENGINE command 182, 183
EQ logical operator 51, 146
  in NEXT WHERE phrase 146
EQ_MASK logical operator in NEXT WHERE phrase 146
error messages 111, 182
  displaying 111
  retrieving from DBMSs 182
escape characters 46, 47
  in character expressions 46
EX command 131
EXCEEDS logical operator 146
  in NEXT WHERE phrase 146
EXEC command 107
EXIT keyword in GOTO command 112
EXIT_WARNING command (Maintain) 133, 134
EXITREPEAT keyword in GOTO command 112
exponentiation operator 29
expressions 27, 28, 29, 34, 45, 49, 50, 52, 53, 118
  Boolean 50
  character 45
  conditional 52, 118
  date and time 34
  default values 53
  limits in Maintain 27, 28
  logical 49
  null values 53
  numeric 29
  relational 50
  extracting substrings 46

F
FALSE value for logical expressions 49, 50
FETCH SQL command equivalent (Maintain) 142
fields (Maintain)
  naming 19
  null values 53, 200
FILE keyword
  in MAINTAIN command 124
FILES keyword in MAINTAIN command 124
flow of control (Maintain) 64, 69, 112, 115, 157, 158, 159, 160, 162
   CALL command 64
   CASE command 69
   GOTO command 112
   IF command 115
   looping 162
   ON MATCH command 157
   ON NEXT command 158
   ON NOMATCH command 158
   ON NONEXT command 159
   PERFORM command (Maintain) 160
   REPEAT command 162
FOCCOMP ddname/file type 130
FocCount variable 109
FocCurrent variable 74, 109
   with COMMIT command 74
FocError variable 109
FocErrorRow variable 110
FocFetch variable 110
FocIndex variable 110
FocMsg 111
FocMsg stack 111
FOCSET command 184
FOCUS data sources 104, 105, 106
   encrypting 104, 105
   restricting existing data sources 106
FOR keyword 79, 87, 118, 142, 146, 171, 193
   in COPY command 79
   in DELETE command 87
   in INCLUDE command 118
   in NEXT command 142, 146
   in REVISE command 171
   in UPDATE command 193
functions (Maintain) 19, 43, 55, 69, 71, 73, 79, 160, 161
   calling using COMPUTE 79
   calling using PERFORM 160, 161
   date-time 43
   defining 69
   naming 19
   passing parameters 71
   return values 73
   Top 73

G

GE logical operator 51

GET keyword in WINFORM command 197
GET_PREMATCH command 188

global variables 86

GOTO command (Maintain) 112, 114, 162
   data source commands and 114
   ENDCASE command and 114
   PERFORM command and 114, 162
GT logical operator 51

H

HIDE keyword in WINFORM command 197

HIGHEST keyword in STACK SORT command 179
IF command (Maintain) 52, 115
  keyword in conditional expressions 52
IMPORT keyword in MODULE command 141
importing modules 141
  restrictions 141
IN logical operator in NEXT WHERE phrase 146
INCLUDE command 118, 121
  adding data 118
  data source position 121
  null values 121
  unique segments 118
INFER
  creating stacks 121
INFER command 121, 122, 123
INTO keyword 65, 79, 107, 122, 124, 127, 142
  in CALL command 65
  in COPY command 79
  in EXEC command 107
  in INFER command 122
  in MAINTAIN command 124
  in MATCH command 127
  in NEXT command 142
IS logical operator in NEXT WHERE phrase 146
IS_NOT logical operator in NEXT WHERE phrase 146
issuing DROP TABLE command 183

K
KEEP keyword 65, 107, 112
  in CALL command 65
  in EXEC command 107
  in GOTO command 112

L
LANGUAGE parameter 184
Language Wizard 55
LE logical operator 51
libraries (Maintain)
  importing 141
  restrictions 141
line feeds in character expressions 46
LOADIMAGE command (Maintain) 134
local variables 86
log files (Maintain) 174, 176, 189
  SAY command 174
  segment and stack values 176
  TYPE command 189
logical expressions 49, 50, 51
  Boolean expressions 50
  evaluating 50
  operators 50, 51
  relational expressions 50
logical operators 50
looping in Maintain language 57
loops 165, 169
  branching out of 169
  ending 169
  simple 165
LT logical operator 51

M
MAINTAIN command 123, 124, 125, 126
  calling a procedure from another procedure 125, 126
  specifying data sources 125
Maintain functions 55, 69, 71, 73, 79, 160, 161
  calling using COMPUTE 79
  calling using PERFORM 160, 161
  defining 69
  passing parameters 71
  return values 73
  Top 73
Maintain language 17, 18, 19, 21, 22, 23, 24, 55, 56, 57, 58, 59, 60, 62
  case sensitivity 18
  class libraries 62
  commands 24, 55, 56, 57
  comments 22, 24
Maintain language (continued)
  conditional actions 60
  displaying forms 57
  encrypting files 57
  function libraries 62
  loops 57
  manipulating stacks 58
  messages and logs 62
  multi-line commands 23
  naming rules 18, 19, 21
  reading data 59
  transferring control 56
  variables 58
  writing transactions 60
Maintain procedures 22, 24, 56, 104, 107, 130
  blank lines 22
  calling 107
  comments 24
  compiling 130
  encrypting 104
  running 56
Master Files 104, 105
  encrypting 104
  performance considerations 105
MATCH command 126, 127, 128, 149, 169
  NEXT command and 149
  REPOSITION command and 169
MATCH keyword in ON MATCH command 157
MESSAGE parameter 184
MISSING attribute 42, 53, 200
  date-time data type and 42
MISSING constant 53, 54
missing data and INCLUDE command 121
MISSING keyword 75, 84
  in COMPUTE command (Maintain) 75
  in DECLARE command (Maintain) 84
MNTCON COMPILE command 130
MNTCON EX command 131
MNTCON EXIT_WARNING command 133, 134
MNTCON LOADIMAGE command 134
MNTCON PERFORMANCE_ANALYSIS command 136
MNTCON RADIO_BUTTON_EMIT_TEXT command 137
MNTCON REMOTESTYLE command 137
MNTCON RUN command 138
MOD operator 30
modular processing 64
MODULE IMPORT command 141
modules 141
  importing 141
  multiplication operator 29
N
names 19
naming conventions 21
  reserved words 21
native-mode arithmetic 31
NE logical operator 51, 146
  in NEXT WHERE phrase 146
NE_Mask logical operator in NEXT WHERE phrase 146
NEEDS keyword 75, 84
  in COMPUTE command 75
  in DECLARE command 84
nested BEGIN blocks 63, 64
nesting IF commands 117
NEXT command (Maintain) 142, 144, 145, 146, 149, 151, 152, 154, 156, 169
  copying data 144
  data source navigation 149, 151, 152, 154
  loading multi-path transaction data 145
  reading data 149
  REPOSITION command 169
  retrieving rows 146
  unique segments 156
  with MATCH command 149
  with multiple rows 146
  with path instances 152
NEXT keyword in ON NEXT command 158
NODATA parameter 53, 184, 200
  null representation 53
NODATA parameter (continued)
setting in a Maintain procedure 184
NOMATCH keyword in ON NOMATCH command 159
NONEXT keyword in ON NONEXT command 160
NOT logical operator 50, 51
NOT_IN logical operator in NEXT WHERE phrase 146
null values 53, 54, 75, 121, 200
  COMPUTE command 75
  expressions 53
  forms 200
  INCLUDE command 121
  testing 54
numeric expressions 29, 30, 31, 32, 33
  Continental Decimal Notation (CDN) 33
  DIV 29, 30
  evaluating 31
  identical operand formats 32
  MOD 30
  operand formats 31, 32
  operators 29
  truncating decimal values 32

O
objects
  creating 57
  declaring 84
OFF keyword 75, 84
  in COMPUTE command 75
  in DECLARE command 84
OMITS logical operator 51, 146
  in NEXT WHERE phrase 146
ON keyword 75, 84, 124, 189
  in COMPUTE command 75
  in DECLARE command 84
  in MAINTAIN command 124
  in TYPE command 189
ON MATCH command 157
ON NEXT command 158
ON NOMATCH command 158, 159
ON NONEXT command 159, 160

ON TABLE SET command 177
operand format for dates 38
OR logical operator 50, 51

P
parameters 71
PASS setting 184, 185
PERFORM command (Maintain) 71, 114, 160, 161, 162
  data source commands and 162
  GOTO command and 114, 162
  nesting 162
PERFORMANCE_ANALYSIS command 136
PRE_MATCH setting 188
presentation logic
  WINFORM command 197
procedures (Maintain) 22, 24, 104, 107, 130
  blank lines 22
  calling 107
  comments 24
  compiling 130
  encrypting 104
properties (Winforms)
  changing dynamically 200

R
RADIO_BUTTON_EMIT_TEXT command (Maintain) 137
records
  selecting 126, 142
REFRESH keyword in WINFORM command 197
relational expressions 50
REMOTE_STYLE command (Maintain) 137
REPEAT command (Maintain) 162, 163, 165, 166, 168, 169
  branching out of loops 169
  ending loops 169
  establishing counters 166
  nested loops 166, 168
REPEAT command (Maintain) (continued)
simple loops 165
UNTIL 166
WHILE 166
REPOSITION command (Maintain) 169, 170
RESET keyword in GOTO command 112
RESET keyword in WINFORM command 197
RESTRICT command 106
return values for functions 73
RETURNS keyword in CASE command 70
REVISE command 170, 171, 172, 173
ROLLBACK command 173, 174
rounding numeric values 32
RUN command 138

S

SAY command 174, 175, 176
choosing between SAY and TYPE commands 176
SEG prefix with SAY command 176
segments 89, 118, 156, 170
DELETE command 89
INCLUDE command 118
NEXT command 156
REVISE command 170
SELECT SQL command equivalent (Maintain) 142
SET command 176, 177
SET keyword in WINFORM command 197
SET parameters 53, 177, 184, 188, 189, 200
NODATA 53, 200
PREMATCH 188, 189
setting in Maintain procedure with
SYS_MGR.FOCSET 184
with ON TABLE 177
SET_PREMATCH setting 188, 189
SHOW keyword in WINFORM command 197
SHOW_ACTIVE keyword in WINFORM command 197
SHOW_AND_EXIT keyword in WINFORM command 197
SHOW_INACTIVE keyword in WINFORM command 197
SOME keyword 75, 84
in COMPUTE command 75
in DECLARE command 84
SORT keyword in STACK SORT command 179
STACK CLEAR command 178, 179
stack commands
creating 121
STACK CLEAR 178
STACK SORT 179
STACK keyword in COPY command 79
STACK prefix with SAY command 176
STACK SORT command 179, 180
stack variables 58, 109, 110
FocCount 109
FocIndex 110
stacks
clearing 178
copying stack rows 79
creating 121
creating stacks with INFER 121
copying stack rows 79
editing 58, 79
naming 19
non-data source columns 123
rows 179
STACK CLEAR command 178
STACK SORT command 179
virtual fields 78
strong concatenation 46
substrings
extracting 47
subtraction in date expressions 39
subtraction operator 29
SYS_MGR global object 181
SYS_MGR.DBMS_ERRORCODE command 182
SYS_MGR.ENGINE command 182, 183
SYS_MGR.FOCSET command 184
SYS_MGR.GET_PREMATCH command 188
SYS_MGR.PRE_MATCH command 188
SYS_MGR.SET_PREMATCH command 188

system variables (Maintain) 58, 59, 60, 109, 110
   FocCurrent 109
   FocError 109
   FocErrorRow 110
   FocFetch 110

T

TAKES keyword in CASE command 70
temporary fields and columns (Maintain)
in data sources 78
text expressions 45
text format 48, 49, 107
   passing variables to procedures 107
   variable length 48, 49
THEN keyword 52, 115
   in conditional expressions 52
   in IF command 115
time expressions 34
TO keyword in SAY command 175

Top function 73, 141
   libraries and 141
TRACEOFF setting 184
TRACEON setting 184
TRACEUSER setting 184

transaction data sources 144
transaction processing
   collecting values 197
   COMMIT command 73
   DELETE command 87
   INCLUDE command 118
   multiple DBMSs rollback 174
   reading multiple-path data sources 145
   REVISE command 170
   ROLLBACK command 173, 174
   UPDATE command 193

transaction variables (Maintain) 59, 60, 109, 110, 196
   FocCurrent 109
   FocError 109
   FocErrorRow 110
   FocFetch 110
   UPDATE command 196
troubleshooting 111
displaying WebFOCUS Server messages 111
TRUE value for logical expressions 49, 50

TX data type 48, 49

TYPE (Maintain) 191, 192

TYPE command 176, 189, 191, 192, 193
   choosing between SAY and TYPE commands 176
   embedding spacing information 191
   including variables in a message 191
   justifying variables 192
   multi-line message strings 192
   truncating 192
   writing information to a file 193

U

UNHIDE keyword in WINFORM command 197
unique segments (Maintain) 89, 118, 156, 197
   DELETE command 89
   INCLUDE command 118
   NEXT command 156
   UPDATE 197
   UPDATE command 197

UNTIL keyword in REPEAT command 163
UPDATE command (Maintain) 193, 194, 195, 196, 197
   data source position 196
   stacks and 196
   transaction variables 196
   unique segments 197

updating data sources 170, 193
   REVISE command 170
   UPDATE command 197
   UPDATE command 193

USER setting 184
V

variable-length character format 65, 107, 124
  passing A0 variables to procedures 124
  passing variables between procedures 65, 107
variable-length strings 48, 49
variables 58, 59, 60, 74, 84, 86
  assigning values with COMPUTE 74
  creating 58
  declaring 74, 84
  local vs. global 86
  system 58, 59, 60
virtual fields in data sources 78

W

WARNING setting 184

weak concatenation 46
WebFOCUS Servers 62, 111, 176, 184
  changing environment settings 176, 184
  displaying messages 111
WHERE keyword 79, 142, 146
  in COPY command 79
  in NEXT command 142, 146
WHILE keyword in REPEAT command 163
WINFORM command 197
wizards 55

Y

YRT keyword 75, 84
  in COMPUTE command 75
  in DECLARE command 84
YRTHRESH setting 184
Customer Connections

Where users partner with Information Builders to provide feedback and create quality documentation.

Partner With Us

Information Builders Technical Content Management team is comprised of many talented individuals who work together to design and deliver quality technical documentation products. Your feedback supports our ongoing efforts!

You can also join a usability study to get an early look at new documentation products and services. Your participation helps us create documentation products that meet your needs.

To send us feedback or make a customer connection, contact Michele Brady, Service Director, Technical Content Management at Michele_Brady@ibi.com.

Request Permission to Use Copyright Material

To request permission to repurpose copyrighted material, please contact Frances Gambino, Executive Director, Technical Content Management at Frances_Gambino@ibi.com.

Visit the Information Builders Technical Documentation Library

To visit the Technical Documentation Library, go to http://documentation.informationbuilders.com/.