

WebFOCUS

WebFOCUS Maintain Language Reference
Version 5 Release 2

Cactus, EDA/SQL, FIDEL, FOCCALC, FOCUS, FOCUS Fusion, FOCUS Vision, Hospital-Trac, Information Builders, the Information Builders logo, Parlay, PC/FOCUS, SmartMart, SmartMode, SNAPPack, TableTalk, WALDO, Web390, WebFOCUS and WorldMART are registered trademarks and EDA, iWay, and iWay Software are trademarks of Information Builders, Inc.

Acrobat and Adobe are registered trademarks of Adobe Systems Incorporated.

AvantGo is a trademark of AvantGo, Inc.

Alpha, DEC, DECnet, and NonStop are registered trademarks and Tru64, OpenVMS, and VMS are trademarks of Compaq Computer Corporation.

CA-ACF2, CA-Datcom, CA-IDMS, CA-Top Secret, and Ingres are registered trademarks of Computer Associates International, Inc.

MODEL 204 and M204 are registered trademarks of Computer Corporation of America.

HP MPE/iX is a registered trademark of Hewlett Packard Corporation.

Intel and Pentium are registered trademarks of Intel Corporation.

ACF/VTAM, AIX, AS/400, CICS, DB2, DRDA, Distributed Relational Database Architecture, IBM, MQSeries, MVS/ESA, OS/2, OS/390, OS/400, RACF, RS/6000, S/390, VM/ESA, VSE/ESA, VTAM, and WebSphere are registered trademarks and DB2/2, HiperSpace, IMS, iSeries, MVS, QMF, SQL/DS, z/OS and z/VM are trademarks of International Business Machines Corporation.

Linux is a registered trademark of Linus Torvalds.

Approach and DataLens are registered trademarks of Lotus Development Corporation.

ObjectView is a trademark of Matesys Corporation.

ActiveX, BizTalk, FrontPage, Microsoft, MS-DOS, PowerPoint, Visual Basic, Visual C++, Visio, Visual FoxPro, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

Teradata is a registered trademark of NCR International, Inc.

Netscape, Netscape FastTrack Server, and Netscape Navigator are registered trademarks of Netscape Communications Corporation.

Nextel is a registered trademark of Nextel Communications.

NetWare and Novell are registered trademarks of Novell, Inc.

Oracle is a registered trademark and Rdb is a trademark of Oracle Corporation.

PeopleSoft is a registered trademark of PeopleSoft, Inc.

INFOAccess is a trademark of Pioneer Systems, Inc.

Progress is a registered trademark of Progress Software Corporation.

BlackBerry is a trademark and RIM is a registered trademark of Research In Motion Limited.

Red Brick Warehouse is a trademark of Red Brick Systems.

Red Hat, RPM, Maximum RPM, Linux Library, PowerTools, Linux Undercover, RHmember, RHmember More, Rough Cuts, and Rawhide are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

SAP and SAP R/3 are registered trademarks and SAP Business Information Warehouse and SAP BW are trademarks of SAP AG.

Siebel is a trademark of Siebel Systems, Inc.

Reliant UNIX is a registered trademark of Fujitsu Siemens Computers.

ADABAS is a registered trademark of Software A.G.

CONNECT:Direct is a trademark of Sterling Commerce.

Java and all Java-based marks, NetDynamics, Solaris, SunOS, and iPlanet are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PowerBuilder, Powersoft, and Sybase are registered trademarks and SQL Server is a trademark of Sybase, Inc.

Unicode is a trademark of Unicode, Inc.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SWIFT is trademark of Synopsys, Inc.

Due to the nature of this material, this document refers to numerous hardware and software products by their trade names. In most, if not all cases, these designations are claimed as trademarks or registered trademarks by their respective companies. It is not this publisher's intent to use any of these names generically. The reader is therefore cautioned to investigate all claimed trademark rights before using any of these names other than to refer to the product described.

Copyright © 2003, by Information Builders, Inc. All rights reserved. This manual, or parts thereof, may not be reproduced in any form without the written permission of Information Builders, Inc.

Printed in the U.S.A.

Preface

This documentation provides a reference of the Maintain language. It is intended for application developers who are responsible for planning the enterprise's software environment and for operating WebFOCUS Maintain. This documentation is part of the WebFOCUS Maintain documentation set.

How This Manual Is Organized

This manual includes the following chapters:

Chapter/Appendix		Contents
1	<i>Language Rules Reference</i>	Describes the rules for using the Maintain language.
2	<i>Command Reference</i>	Describes the purpose, syntax, and usage notes of all Maintain commands and system variables.
3	<i>Expressions Reference</i>	Describes the rules for combining variables, constants, operators, and functions to produce different types of expressions.
4	<i>Form and Control Properties Reference</i>	Describes all Maintain form and control properties.
A	<i>WebFOCUS Maintain Error Messages</i>	Lists all of WebFOCUS Maintain's error messages.

About Your Documentation

WebFOCUS Maintain documentation includes one printed manual, a context-sensitive help system, and three corresponding manuals that are provided as PDF files. The manuals provided are:

- *WebFOCUS Maintain Getting Started* provides an introduction to WebFOCUS Maintain. It includes an overview of how to develop applications and a discussion of WebFOCUS Maintain concepts. A step-by-step tutorial is provided to prepare you for your first steps with the Maintain Development Environment. (Print and PDF.)
- *Developing WebFOCUS Maintain Applications* describes how to develop and deploy WebFOCUS Maintain applications using the Maintain Development Environment (PDF only).

- *WebFOCUS Maintain Language Reference* which is this manual. (PDF only.)

Since WebFOCUS Maintain is part of the WebFOCUS Developer Studio suite of tools, you will also need to have available your WebFOCUS Developer Studio documentation.

Documentation Conventions

The following conventions apply throughout this manual:

Convention	Description
THIS TYPEFACE or <i>this typeface</i>	Denotes syntax that you must enter exactly as shown.
<i>this typeface</i>	Represents a placeholder (or variable) in syntax for a value that you or the system must supply.
<u>underscore</u>	Indicates a default setting.
<i>this typeface</i>	Represents a placeholder (or variable) in a text paragraph, a cross-reference, or an important term.
this typeface	Highlights a file name or command in a text paragraph that must be lowercase.
<i>this typeface</i>	Indicates a button, menu item, or dialog box option you can click or select.
Key + Key	Indicates keys that you must press simultaneously.
{ }	Indicates two or three choices; type one of them, not the braces.
[]	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 points (...).
.	Indicates that there are (or could be) intervening or additional commands.

Related Publications

Visit our World Wide Web site, <http://www.informationbuilders.com>, to view a current listing of our publications and to place an order. You can also contact the Publications Order Department at (800) 969-4636.

Customer Support

Do you have questions about WebFOCUS Maintain?

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

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

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

Information You Should Have

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

- Your six-digit site code (xxxx.xx).
- Your WebFOCUS configuration:
 - The front-end you are using, including vendor and release.
 - The communications protocol (for example, TCP/IP or HLLAPI), including vendor and release.
 - The software release.
 - The server you are accessing, including release (for example, 5.2).

- The stored procedure (preferably with line numbers) or FOCUS commands being used in server access.
- The name of the Master File and Access File.
- The exact nature of the problem:
 - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
 - The error message and return code, if applicable.
 - Is this related to any other problem?
- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?
- What release of the operating system are you using? Has it, WebFOCUS, 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 the code to access a single data source?
- Do you have a trace file?
- How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

User Feedback

In an effort to produce effective documentation, the Documentation Services staff welcomes any opinion you can offer regarding this manual. Please use the Reader Comments form at the end of this manual to relay suggestions for improving the publication or to alert us to corrections. You can also use the Documentation Feedback form on our Web site, <http://www.informationbuilders.com>.

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 World Wide Web site (<http://www.informationbuilders.com>) or call (800) 969-INFO to speak to an Education Representative.

Contents

1. Language Rules Reference	1-1
Case Sensitivity	1-2
Specifying Names	1-2
Reserved Words	1-4
What Can You Include in a Procedure?	1-6
Multi-line Commands	1-6
Terminating a Command's Syntax	1-7
Adding Comments	1-7
2. Command Reference	2-1
Language Summary	2-2
Defining a Procedure	2-2
Defining a Maintain Function	2-2
Blocks of Code	2-2
Transferring Control	2-2
Executing Procedures	2-3
Encrypting Files	2-3
Loops	2-4
Forms	2-4
Defining Classes	2-4
Creating Variables	2-4
Assigning Values	2-4
Manipulating Stacks	2-4
Selecting and Reading Records	2-5
Conditional Actions	2-6
Writing Transactions	2-6
Setting WebFOCUS Server Parameters	2-7
Using Libraries of Classes and Functions	2-7
Messages and Logs	2-8
BEGIN	2-8
CALL	2-10
CASE	2-14
Calling a Function: Flow of Control	2-16
Passing Parameters to a Function	2-16
Using a Function's Return Value	2-17
The Top Function	2-17
COMMIT	2-18
COMPILE	2-19
COMPUTE	2-20
Using COMPUTE to Call Functions	2-24

Contents

COPY	2-24
DECLARE	2-28
Local and Global Declarations	2-30
DECRYPT	2-31
DELETE	2-31
DESCRIBE	2-34
ENCRYPT / DECRYPT	2-36
Encrypting Procedures	2-37
Encrypting Data	2-37
Performance Considerations	2-37
Restricting Existing Data Sources	2-38
END	2-39
EXEC	2-39
FocCount	2-43
FocCurrent	2-43
FocError	2-43
FocErrorRow	2-44
FocFetch	2-44
FocIndex	2-44
FocMsg	2-45
GOTO	2-46
Using GOTO With Data Source Commands	2-48
GOTO and ENDCASE	2-49
GOTO and PERFORM	2-49
IF	2-50
Coding Conditional COMPUTE Commands	2-52
INCLUDE	2-53
Data Source Position	2-56
Null Values	2-56
INFER	2-56
Defining Non-Data Source Columns	2-58
MAINTAIN	2-59
Specifying Data Sources With the MAINTAIN Command	2-60
Calling a Procedure From Another Procedure	2-61
MATCH	2-62
How the MATCH Command Works	2-64
MODULE	2-64
What You Can and Cannot Include in a Library	2-65

NEXT	2-65
Copying Data Between Data Sources	2-68
Loading Multi-Path Transaction Data	2-69
Retrieving Multiple Rows: The FOR Phrase	2-69
Using Selection Logic to Retrieve Rows	2-70
NEXT After a MATCH	2-70
Data Source Navigation Using NEXT: Overview	2-71
Data Source Navigation: NEXT With One Segment	2-72
Data Source Navigation: NEXT With Multiple Segments	2-73
Data Source Navigation: NEXT Following NEXT or MATCH	2-75
Unique Segments	2-77
ON MATCH	2-78
ON NEXT	2-79
ON NOMATCH	2-80
ON NONEXT	2-81
PERFORM	2-82
Using PERFORM to Call Maintain Functions	2-82
Using PERFORM With Data Source Commands	2-83
Nesting PERFORM Commands	2-83
Avoiding GOTO With PERFORM	2-83
REPEAT	2-83
Branching Within a Loop	2-90
REPOSITION	2-90
REVISE	2-91
ROLLBACK	2-94
DBMS Combinations	2-95
RUN	2-96
SAY	2-97
Writing Segment and Stack Values	2-98
Choosing Between the SAY and TYPE Commands	2-98
STACK CLEAR	2-98
STACK SORT	2-99
SYS_MGR	2-100
Deactivating Preliminary Database Operation Checking With SYS_MGR.PREMATCH	2-101
Retrieving DBMS Codes With SYS_MGR.DBMS_ERRORCODE	2-102
Issuing Native SQL Commands (SQL Passthru):	2-102
Setting System Values for a WebFOCUS Server With SYS_MGR.FOCSET	2-104

TYPE	2-105
Including Variables in a Message	2-107
Embedding Horizontal Spacing Information	2-107
Embedding Vertical Spacing Information	2-107
Coding Multi-Line Message Strings	2-108
Justifying Variables and Truncating Spaces	2-108
Writing Information to a File	2-108
UPDATE	2-109
Update and Transaction Variables	2-111
Data Source Position	2-112
Unique Segments	2-112
WINFORM	2-113
Displaying Default Values in a Form	2-116
Dynamically Changing Form Control Properties	2-116
3. Expressions Reference	3-1
Types of Expressions You Can Write	3-2
Expressions and Variable Formats	3-3
Writing Numeric Expressions	3-3
Order of Evaluation	3-5
Evaluating Numeric Expressions	3-6
Identical Operand Formats	3-6
Different Operand Formats	3-7
Continental Decimal Notation	3-7
Writing Date and Time Expressions	3-8
Formats for Date Values	3-8
Evaluating Date Expressions	3-9
Selecting the Format of the Result Variable	3-10
Manipulating Dates in Date Format	3-11
Using a Date Constant in an Expression	3-11
Extracting a Date Component	3-11
Combining Variables With Different Components in an Expression	3-12
Different Operand Date Formats	3-12
Using Addition and Subtraction in a Date Expression	3-13
Specifying Date-Time Values	3-14
Describing Date-Time Values	3-16
Date-Time Display Formats	3-17
ACTUAL Formats for Date-Time Values	3-22
Manipulating Date-Time Values Directly	3-23
Comparison and Assignment	3-24
Date-Time Subroutines	3-25
Manipulating Date-Time Values With Subroutines	3-27

Writing Character Expressions	3-37
Concatenating Character Strings	3-38
Evaluating Character Expressions	3-38
Variable-Length Character Variables	3-40
Writing Logical Expressions	3-41
Relational Expressions	3-42
Boolean Expressions	3-42
Evaluating Logical Expressions	3-42
Writing Conditional Expressions	3-44
Handling Null Values in Expressions	3-44
Assigning Null Values: the MISSING Constant	3-45
Conversion in Mixed-Format Null Expressions	3-45
Testing Null Values	3-45
4. Form and Control Properties Reference	4-1
(GroupCode) Property	4-2
(Name) Property	4-2
Alignment Property	4-2
AlternateRowColor Property	4-3
BackColor Property	4-3
BackColorOver Property	4-4
BackgroundImage Property	4-4
Border Property	4-5
BorderColor Property	4-6
BorderText Property	4-6
BorderWidth Property	4-6
Bottom Property	4-7
CaseStyle Property	4-7
Checked Property	4-7
Columns Property	4-8
Content Property	4-8
CursorPointer Property	4-8
DefaultButton Property	4-9
Enabled Property	4-10
FixedColumns Property	4-10
Font Property	4-11
ForeColor Property	4-11
ForeColorOver Property	4-12
GridLines Property	4-12
HeaderBackColor Property	4-12
HeaderFont Property	4-13
HeaderForeColor Property	4-13
Headers Property	4-14

Contents

Height Property	4-14
Help Property	4-15
Hyperlink Property	4-15
Image Property	4-16
ImageDown Property	4-16
ImageOver Property	4-16
ItemBorder Property	4-16
Layer Property	4-17
Left Property	4-17
ListItems Property	4-17
Map Property	4-17
MaximizeBox and MinimizeBox Properties	4-18
MultiSelection Property	4-18
Orientation Property	4-19
Overflow Property	4-19
Password Property	4-20
PenWidth Property	4-20
ReadOnly Property	4-20
Right Property	4-21
Rows Property	4-21
ScrollBars Property	4-21
ScrollHeight and ScrollWidth Properties	4-22
Scrolling Property	4-22
SelectedItem/SelectedItems Property	4-23
Sizeable Property	4-23
Source Property	4-24
Stretched Property	4-24
Tabstop Property	4-24
Text Property	4-25
TextOnLeft Property	4-25
Title Property	4-26
ToolTipText Property	4-26
Top Property	4-26
Validation Property	4-27
Visible Property	4-27
Width Property	4-27
A. WebFOCUS Maintain Error Messages	A-1

CHAPTER 1

Language Rules Reference

Topics:

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

You can use the Maintain language more effectively if you are familiar with its rules:

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

Data source descriptions and external 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 external procedures depend on what type of procedure it is and what logic it contains.

Case Sensitivity

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. The only two exceptions are the MAINTAIN and END keywords used to begin and end a request: these must be in uppercase.

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

REPEAT

repeat

RePeat

REPeat

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 any case.

While 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.'

Specifying Names

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

- **Length of names.** Unqualified names that are defined in a Maintain procedure—such as the unqualified names of forms, cases, 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 do 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 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, virtual servers, and deployment scenarios can also include embedded spaces. (Other types of names cannot include spaces.)

- **Identical names.** Most types of items in a WebFOCUS Maintain project can have the same name. The only exceptions are data sources, stacks, and forms, which cannot have the same name within the same Maintain procedure.

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

- **Qualified names.** In general, whenever a name can be qualified, 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 reader 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 form 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 an object's member functions and member variables, you should always use the function's or variable's fully-qualified name (that is, the name in the form *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; when 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; and when different forms 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) . datasourcenam e . 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 out 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 a field's ALIAS attribute in a Master File.)

Reserved Words

The words in the following table 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 forms.

In addition to these words, you may not use the names of built-in functions to name functions that you create yourself. See the *WebFOCUS 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.

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

What Can You Include in a Procedure?

You can include the following items in a Maintain procedure:

- **Maintain language commands**, which are described in Chapter 2, *Command Reference*.

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 1-7. 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 an application's starting procedure (sometimes known as a root procedure), and it is not called by any other Maintain procedures, it can also contain EDA Dialogue Manager commands preceding the MAINTAIN command. Dialogue Manager commands are described in the iWay documentation.

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 reader clarity:

```
MAINTAIN FILES VideoTrk
    AND Movies
.
.
.
IF CustInfo.FocIndex GT 1
    THEN COMPUTE CustInfo.FocIndex = CustInfo.FocIndex - 1;
    ELSE COMPUTE CustInfo.FocIndex = CustInfo.FocCount;
```

Terminating a Command's 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, it is recommended that you supply it.

Coding suggestion: One reason why it is preferable to supply optional semicolons is that 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. Another reason is that if you supply optional semicolons 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 command's version of the 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 Chapter 2, *Command Reference*.

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: on its own line, at the end of a command, or even in the middle of a command; 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
```

Note that 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.

Adding Comments

CHAPTER 2

Command Reference

Topic:

- Language Summary

This reference provides a summary of the Maintain language's 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

The following command defines Maintain functions:

CASE

Defines a Maintain function. Maintain functions are also known as cases.

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 allow transfer of 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 execute procedures or prepare them for execution:

CALL

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

EXEC

Executes an external procedure.

COMPILE

Compiles a procedure to increase its execution speed. This command is outside the Maintain language, but is described here in *Command Reference* for your convenience.

RUN

Executes a compiled procedure. This command is outside the Maintain language, but is described here in *Command Reference* for your convenience.

Encrypting Files

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

ENCRYPT / DECRYPT

Encodes procedure files and Master Files to prevent unauthorized users from viewing their contents. This command is outside the Maintain language, but is described here in *Command Reference* for your convenience.

DECRYPT

Decodes files that have been encoded using the ENCRYPT command. This command is outside the Maintain language, but is described here in *Command Reference* for your convenience.

Loops

The following command supports looping:

`REPEAT`

Enables a circular flow of control.

Forms

The following command is responsible for presentation logic:

`WINFORM`

Displays a form by which application end users can read, enter, and edit data, and manipulates 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, including objects:

`DECLARE`

Creates local and global variables.

`COMPUTE`

Creates global variables. 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 4th 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 count of 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 changes 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 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.

In addition, you can use the following commands to directly interface with a DBMS:

`SYS_MGR.SET_PREMATCH`

Turns off preliminary database operation checking before an update.

`SYS_MGR.GET_PREMATCH`

Determines whether prematch checking is on or off.

`SYS_MGR.ENGINE`

Pass 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 fails.

Writing Transactions

The commands which can be used to control transactions are:

`INCLUDE`

Adds one or more new data source records.

`UPDATE`

Updates the specified data source fields. 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.

In addition, 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.

In addition, you can use the following commands to directly interface with a DBMS:

SYS_MGR.SET_PREMATCH

Turns off preliminary database operation checking before an update.

SYS_MGR.GET_PREMATCH

Determines whether prematch checking is on or off.

SYS_MGR.ENGINE

Pass 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 command:

SYS_MGR.FOCSET

Sets WebFOCUS Server parameters (without having to set them in EDASPROF).

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 external 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

BEGIN Command

The syntax for the BEGIN command is

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

where:

BEGIN

Specifies the start of a BEGIN/ENDBEGIN 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.

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.

Example BEGIN in 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 in ON NEXT

This example shows BEGIN and ENDBEGIN with ON NEXT:

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

Example BEGIN in IF

You can also use BEGIN and ENDBEGIN with IF to execute 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 execute a series of commands when the prior command fails:

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

CALL

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:

```
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
```

CALL

Use the CALL command when you need one procedure to call another. When you use CALL, both the calling and called procedures communicate via variables: local variables that you pass between them and the global transaction variable FocError. 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 *WebFOCUS Maintain Concepts in Getting Started*.

Syntax CALL Command

The syntax of the CALL command is

```
CALL procedure [AT server] [KEEP|DROP] [FROM var_list] [INTO var_list]
[;]
```

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

where:

procedure

Is the name of the Maintain procedure to execute.

AT *server*

Identifies the called procedure's server if the called procedure and calling procedure are on different servers. In some situations the Maintain Development Environment supplies this phrase and you should refrain from coding it yourself. See *WebFOCUS Maintain Concepts in WebFOCUS Maintain Getting Started* for more information.

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 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 A0 or TX) and those defined using STACK OF.

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.

If you specify KEEP or DROP but the called procedure is not on a different server, Maintain simply ignores the KEEP or DROP keyword.

;

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 Chapter 1, *Language Rules Reference*.

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 redisplay 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:

```

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:

```

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

```

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/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;
WINFORM 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 function's beginning, and the ENDCASE keyword defines its end.

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

You can call a Maintain function by issuing a PERFORM or GOTO command, calling the function directly, triggering the function as an event handler, or calling it via 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

CASE Command

The syntax for the CASE command is

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

where:

functionname

Is the name you give to the function, and 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 function's parameters (*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. See *Defining Events and Event Handlers in Developing WebFOCUS Maintain Applications* for information about event handlers.

RETURNS *result/t*

Specifies that the function returns a value. *result* is the name of the variable being returned, and *t* is the variable's data type. 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. See *Defining Events and Event Handlers in Developing WebFOCUS Maintain Applications* for information about event handlers.

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 CASE command's parameter and return variable definitions. 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 Chapter 1, *Language Rules Reference*.

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 reached the end of that function, it returns to the command following PERFORM. See *PERFORM* on page 2-82.
- **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. See *GOTO* on page 2-46.

Calling a Function: Flow of Control

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

After the Increase function completes in the following example, 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, a Maintain function's parameters are both input and output parameters:

- When one function calls another, the calling function passes the parameters' current values.
- When the called function terminates, it passes the parameters' current values back.

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.

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 parameter's value is converted to the new data type. (Data types, in this context, refer to basic data types: fixed-length character—that is, A_n where n is greater than zero; variable-length character—that is, $A0$ and 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 parameter's value 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 parameter variable's value, 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 a Function's Return Value

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

```

MAINTAIN FILE HousePlan
.
.
.
CASE FindArea TAKES Length/D6.2, Width/D6.2 RETURNS Area/D6.2;
Area = Length * Width;
ENDCASE
.
.
.
COMPUTE ConferenceRoom/D6.2 = FindArea(CRlength,CRwidth);
.
.
.
END

```

The Top Function

When you execute 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:

- **Explicitly**, 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

- **Implicitly**, without a CASE Top command. In the absence of CASE Top, Maintain assumes that there is an implied Top function that:
 - Begins with the first executable command that is outside of a function (that is, outside of a CASE command).
 - Ends with the next non-executable command.

For the purpose of this description, Maintain considers nonexecutable commands to be CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE; all other commands are considered to be executable.

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 transaction's 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

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 Chapter 1, *Language Rules Reference*.

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, it means that the COMMIT failed and all of the records in the logical unit of work will be rolled back because an internal ROLLBACK is issued.

COMPILE

The COMPILE command creates a compiled procedure which, under:

- **Windows NT and UNIX** has an extension of .FCM.
- **OS/390** is allocated to ddname FOCCOMP.
- **CMS** has a file type of FOCCOMP.

You can reduce the time needed to start a procedure that contains forms by compiling the procedure. The more frequently the 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. You can issue this command from within an external procedure, not from within a Maintain procedure.

Syntax **COMPILE Command**

The syntax of the COMPILE command is

```
COMPILE procedure_name [AS newname]
```

where:

procedure_name

Is the name of the uncompiled procedure.

newname

Is the name given to the new compiled procedure file. If you do not supply a name, the name of the compiled procedure defaults to the name of the uncompiled procedure.

Reference **Commands Related to COMPILE**

RUN executes compiled procedures.

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 2-30 for more information about local and global variables.)
- Assign a value to an existing variable.

Syntax

COMPUTE Command

The syntax of the COMPUTE command is

```
[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 creating a simple variable, you can specify all Maintain built-in formats and edit options (except for TX) as described for the Master File FORMAT attribute in *Describing Data With the WebFOCUS Language*; if creating an object, you can specify a class. You must specify a data type when you create a new variable. You can only specify a variable's data type once, and you cannot redefine an existing variable's data type.

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 *WebFOCUS Developing Reporting Applications*.

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. For information about working with cross-century dates, see *WebFOCUS Developing Reporting Applications*.

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 *WebFOCUS Developing Reporting Applications*.

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 variable's default value 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.

COMPUTE

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.

=

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 Chapter 3, *Expressions Reference*. 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 = 00000000;
```

is the same as:

```
COMPUTE TempEmp_ID/A9 = 00000000;
```

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 so long as each expression is ended with a semicolon. For example:

```
COMPUTE TempEmp_ID/A9 = 00000000;  
TempLast_Name/A15 ;  
TempFirst_Name/A10;
```

Example Combining Several Statements Onto 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=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 field's segment.
- **DECLARE** creates local and global variables. See *DECLARE* on page 2-28.

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:

```
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:

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

For more information about terminating commands with semicolons, see *Terminating a Command's Syntax* in Chapter 1, *Language Rules Reference*.

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.

The COPY command 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 **COPY Command**

The syntax of the COPY command is

```
[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 stack into the target 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 stack into the target 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 which 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 Chapter 1, *Language Rules Reference*.

Example Copying All Rows of a Stack

The following 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 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 copies the first row of the Emp stack into the first row in the Temp stack. Only the first row in the source 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 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

- To copy an entire stack, specify FOR ALL without a subscripted source stack.
- Stack columns created using the COMPUTE command cannot be copied into the Current Area.
- Source stack rows will overwrite the specified target stack rows if they already exist.
- If the COPY command creates rows in the target stack, and the target stack contains columns that are not in the source stack, those columns in the new rows will be initialized to their default values of blank, zero, or null (missing).
- If the source stack has more columns than the target stack, only corresponding columns are copied.
- The FOR prefix copies rows from the source stack one row at a time, not all at the same time. For example, the following command

```
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 so on. 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 2-30 for more information.

Syntax **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 Maintain language's standard naming rules; for more information, see *Specifying Names* in Chapter 1, *Language Rules Reference*.

datatype

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

expression

Is an optional expression that will provide the variable's initial value. If the expression is omitted, the variable's initial value 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 *WebFOCUS Developing Reporting Applications*.

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 *WebFOCUS Developing Reporting Applications*.

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 variable's default value 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.

DECLARE

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. See *DESCRIBE* on page 2-34.
- **COMPUTE** creates global variables (including objects) and assigns values to existing variables. See *COMPUTE* on page 2-20.

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.

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.

DECRYPT

See *ENCRYPT / DECRYPT* on page 2-36.

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 and, when it finds a match, deletes that anchor instance and all its descendants.

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

Syntax

DELETE Command

The syntax of the DELETE command is

```
[FOR {int|ALL}] 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.

DELETE

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 Chapter 1, *Language Rules Reference*.

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 3 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 Winform. If the employee is already in the data source, all records for that employee are deleted from the data source. This includes the employee's instance in the root segment and all descendent instances (such as pay dates, addresses, etc.). In order to find out if the employee is in the data source, a MATCH command is issued:

```
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 if 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 FILE Employee
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 a unique segment's fields 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 at COMMIT time there is a concurrency conflict, the transaction variable FocCurrent is set to a non-zero value.

DESCRIBE

- A DELETE command cannot have more than one input (FROM) stack.
- After a DELETE command completes, 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 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. See *COMMIT* on page 2-18.
- **ROLLBACK** cancels all data sources changes made since the last COMMIT. See *ROLLBACK* on page 2-94.

DESCRIBE

The DESCRIBE command enables you to define classes and to create synonyms for data types.

Syntax DESCRIBE Command

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

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

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

[memfunction

[memfunction] ...

ENDDESCRIBE]
```

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

```
DESCRIBE synonym = datatype ;
```

where:

classname

Is the name of the class that you are defining. The name is subject to the Maintain language's standard naming rules; for more information, see *Specifying Names* in Chapter 1, *Language Rules Reference*.

superclass

Is the name of the superclass from which you wish to derive this class. Include this only if this is to be a subclass.

memvar

Names one of the class's member variables. The name is subject to the Maintain language's standard naming rules; for more information, see *Specifying Names* in Chapter 1, *Language Rules Reference*.

type

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

memfunction

Defines one of the class's member functions. Member functions are defined the same way as other Maintain functions, using the CASE command; see CASE on page 2-14 for more information.

synonym

Is a synonym for a data type (a class or format). The synonym is subject to the Maintain language's standard naming rules; for more information, see *Specifying Names* in Chapter 1, *Language Rules Reference*.

;

For class definitions, this terminates the definition if the definition omits member functions. If it includes member functions, the semicolon is omitted and the ENDDESCRIBE 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 ENDDESCRIBE command must also be omitted, and the definition must be terminated 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 people's names, and defines them all as A30, you would define a data type synonym for A30:

```
DESCRIBE NameType = A30;
```

You would then define all of the name variables as NameType:

```
DECLARE UserName/NameType;
```

```
DECLARE ManagerName/NameType;
```

```
DECLARE CustomerName/NameType;
```

If you needed to change all name variables to A/40, you could change all of them at once simply by changing one data type synonym:

```
DESCRIBE NameType = A40;
```

Example Defining a Class

The following DESCRIBE command defines a class named Floor in an architecture application:

```
DESCRIBE Floor = (Length/I4, Width/I4, Area/I4)
CASE PrintFloor
SAY "length=" Length " width=" Width " area=" Area "\n";
ENDCASE
ENDDESCRIBE
```

Reference Commands Related to DESCRIBE

- **DECLARE** creates local and global variables, including objects. See *DECLARE* on page 2-28.
- **COMPUTE** creates global variables, including global objects, and assigns values to existing variables. See *COMPUTE* on page 2-20.

ENCRYPT / DECRYPT

These commands are outside the Maintain language, but are described here in *Command Reference* for your convenience. You can issue these commands from within an external 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:

```
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:

```
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 file's 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, SEGTYPE=S1, ENCRYPT=ON, $
```

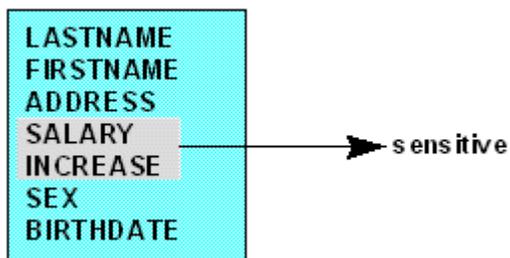
You must specify the ENCRYPT attribute when the file is new, before it contains any data. Encryption cannot be requested later by a change to the Master File and cannot be removed once it has been requested and any data has been entered in the file.

Note: Encryption is used only with FOCUS data sources.

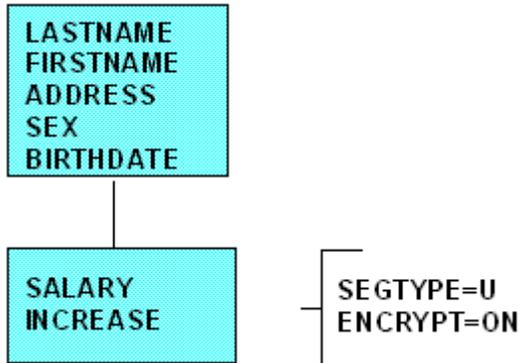
Performance Considerations

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 SEGTYPE=U beneath their original segment.

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



They should be grouped as:



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's 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 **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. See *MAINTAIN* on page 2-59.
- **CALL** is used to call one procedure from another. See *CALL* on page 2-10.

EXEC

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

Syntax EXEC Command

The syntax of the EXEC command is

```
EXEC progrname AT server [KEEP|DROP] [FROM var_list] [INTO stacks] [;]
```

where:

progrname

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

AT *server*

Identifies the called procedure's server. In some situations the Maintain Development Environment supplies this phrase and you should refrain from coding it yourself; see *Getting Started* for more information.

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.

If you specify KEEP or DROP but the called procedure is not on a different server, Maintain simply ignores the KEEP or DROP keyword.

;

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 Chapter 1, *Language Rules Reference*.

Example Retrieving Data From a FOCUS Report

This example retrieves data from a FOCUS report that is run as 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. ENDREPEAT
11. END

```

External procedure GETMOVIE

```

1.  TABLE FILE MOVIES
2.  PRINT MOVIECODE TITLE
3.  ON TABLE PCHOLD
4.  END

```

The following numbers correspond to the line numbers in the client procedure and represent the sequence that the procedure is executed.

1. The MOVIES data source will be used in Line 2 for defining the EXEC data stack with fields from the MOVIES data source.
2. The INFER command defines the MoviesInfo stack with two columns, MOVIECODE and TITLE.
3. The EXEC command is used to execute the external procedure called GETMOVIE on the server CACSERVE. The answer set returned by GETMOVIE populates the stack MoviesInfo.

4. COMPUTE computes the variable I to a value of 1.
5. The REPEAT/ENDREPEAT command is used to cycle through the data stack, MoviesInfo, three times.
6. This multi-line TYPE command displays values from the MoviesInfo stack.
7. Increments the variable I by 1.
8. Defines the bottom of the loop begun in line 5.
9. END ends the procedure.

The external procedure sequence executes as follows:

1. Begins a report request using the MOVIES data source.
2. Creates a two-column report and downloads it to the client procedure via the MoviesInfo stack.
3. END ends the remote procedure.

Example **Passing Variables to an External Procedure**

EXEC allows you to pass scalar information to the external procedure. The previous example retrieves all MovieCodes and Titles. The following example assumes that you only want to retrieve the Dramas. The client procedure would be the following:

Client Procedure

```

MAINTAIN FILE MOVIES
INFER MOVIECODE TITLE INTO MoviesInfo;
COMPUTE Category = 'DRAMA';
EXEC GETMOVIE FROM Category INTO MoviesInfo;
COMPUTE I/I4=1;
REPEAT 3;
    TYPE "MOVIE CODE IS:  <<MoviesInfo(I).MOVIECODE";
    TYPE "          TITLE IS:  <<MoviesInfo(I).TITLE";
ENDREPEAT
END

```

External procedure GETMOVIE

```

TABLE FILE MOVIES
PRINT MOVIECODE TITLE
WHERE CATEGORY EQ '&1'
ON TABLE PCHOLD
END

```

The scalar field CATEGORY is passed to the external procedure. The amper variable in that program is replaced by the value 'Drama' and only the matching records are returned. You can send many scalar variables to the external procedure, and they are received as numbered amper variables.

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 EmplInfo stack:

```
EmplInfo.FocCount
```

The FocCount variable is useful as a test to see whether or not 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 that will 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 = 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:

```
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

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

The fifteenth row of the stack is displayed in the form.

FocMsg

FocMsg is a system stack with one A80 column named Msg. When a Maintain procedure executes either

- An external procedure
- A Maintain procedure on a remote server (that is, a Maintain procedure called via the CALL *procname* AT command)

All of the messages that the called procedure writes to the default output device are automatically copied to the calling Maintain procedure's FocMsg stack. 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:

```
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. ENDREPEAT
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. ENDREPEAT
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 **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. See *GOTO* on page 2-46 and *PERFORM* on page 2-82 for more information.

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 2-14.

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 2-83.

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 2-83.

;

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 Chapter 1, *Language Rules Reference*.

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 2-18.

Reference Commands Related to GOTO

- **PERFORM** transfers control to another function. When the function finishes, control is returned to the command following the PERFORM. See *PERFORM* on page 2-82.
- **CASE/ENDCASE** allows a set of commands to be grouped together. See *CASE* on page 2-14.
- **REPEAT/ENDREPEAT** provides a general looping facility. See *REPEAT* on page 2-83.

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 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 have a command indicating where control should pass to next. 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 2-82 for more information.

GOTO and PERFORM

It is recommended that you do not issue a GOTO command within the scope of a PERFORM command.

A PERFORM command's scope 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 invoke Case One, Case One issues a PERFORM command to invoke Case Two, Case Two issues a PERFORM command to invoke 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 runtime 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 **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. See Chapter 3, *Expressions Reference* for more information about Boolean expressions.

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. See Chapter 3, *Expressions Reference* for additional information.

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. OMTS CONTAINS
7. AND
8. OR

maint_command

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:

```
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";
```

Of course this example could be executed much more efficiently by issuing the following command:

```
TYPE "DEPT EQ <Dept>";
```

You can also use the BEGIN command to place another IF within a THEN phrase. For example:

```

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
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 C221;
    IF C EQ 3 THEN PERFORM C231;
  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/I6 = 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 anchor's ancestor segments, or provide ancestor segment key values in the source stack. This ensures that INCLUDE can navigate from the root to the anchor segment's first instance.

Syntax

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.

INCLUDE

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, 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 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 benefits of including the semicolon, see Chapter 1, *Language Rules Reference*.

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 indicates to start in the second row of the stack and the FOR 3 indicates to include the next 3 rows.

```
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:

```
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 are input without a prior confirmation that they do not already exist in the data source, it is possible that one or more 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 of the rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work and 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. After the INCLUDE completes, the variable FocError is checked to see if the INCLUDE was successful. If it failed, a general error handling function is called:

```
FOR ALL INCLUDE Emp_ID Dat_Inc FROM EmpInfo;
IF FocError NE 0 THEN PERFORM Errhandle;
```

Example Adding Data From the Current Area

This example shows using data from the Current Area. The user is prompted for the employee's 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:

```
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.
- After an INCLUDE command completes, 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 all data source changes since the last COMMIT permanent. See *COMMIT* on page 2-18.
- **ROLLBACK** cancels all data sources changes made since the last COMMIT. See *ROLLBACK* on page 2-94.

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, you 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 need to issue the INFER command to define the stack's structure. 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 data source's Master File provides the columns' names and formats. 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 procedure's MAINTAIN command.
- Have access to the Master File.

Syntax

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, 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 Chapter 1, *Language Rules Reference*.

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;
.
.
.
END
```

Reference Commands Related to Infer

- **CALL** is used to call one Maintain procedure from another. See *CALL* on page 2-10.
- **COPY** can be used to copy data from one stack to another. See *COPY* on page 2-24.
- **COMPUTE** can be used to define the contents of a stack for non-data source fields. See *COMPUTE* on page 2-20.

Defining Non-Data Source Columns

To define stack columns in a procedure for non-data source fields—that is, 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 **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 a stack's data source columns—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.

MAINTAIN

,

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 16 Master Files per MAINTAIN command. If you need to access more than 16 data sources, you can call other procedures that can each access an additional 16 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. See *END* on page 2-39.
- **CALL** is used to call one procedure from another. See *CALL* on page 2-10.

Specifying Data Sources With the MAINTAIN Command

The MAINTAIN command does not require that any parameters be supplied; that is, 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 completes, control returns to the item that follows the CALL command and the calling procedure.

Called procedures can also reside on remote WebFOCUS Servers, allowing you to partition the logic of your application between machines.

For additional information about calling one procedure from another, see *Executing Other Maintain Procedures* in *WebFOCUS Maintain Concepts* in *Getting Started* and information about the CALL command and CALL on page 2-10.

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 completes, 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:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Emps;
INFER emp_id into VALIDATE;

CALL Validate FROM Emps INTO ValidEmps;
.
.
.
END
```

The called procedure (Validate) contains:

```
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 anchor's ancestor segments. 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 command's WHERE phrase.

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 command's WHERE phrase.

Syntax

MATCH Command

The syntax of the MATCH command is

```
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; 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.

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 Chapter 1, *Language Rules Reference*.

Example Matching Keys in the Employee Data Source

The following example performs a MATCH on the key field in the PayInfo segment. It gets its value of Pay_Date from the field Pay_Date 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 EmplInfo segment. It gets the value of 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 EmplInfo 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 EmplInfo 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. See *NEXT* on page 2-65.
- **REPOSITION** changes data source position to be at the beginning of the chain. See *REPOSITION* on page 2-90.

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, it 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.

MODULE

The MODULE command accesses a source code library so that the current procedure can use the library's class definitions and Maintain functions. (A library is a nonexecutable procedure, and is implemented as a project component called an import module.)

Syntax **MODULE Command**

The MODULE command must immediately follow the procedure's 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. See *DESCRIBE* on page 2-34.
- **CASE** defines a function; you can use CASE to include functions in a library. See *CASE* on page 2-14.

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 a just 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 anchor's ancestor segments. 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 anchor's ancestor segments. This enables `NEXT` to navigate from the root to the anchor segment's current instance.

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. For those who 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 **NEXT Command**

The syntax of the NEXT command is

```
[FOR {int|ALL}] NEXT path [INTO stack[(row)]] [WHERE expr [AND expr ...]]
[;]
```

expr: *operand comparison_op operand*

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.

expr

Is any valid NEXT WHERE expression as defined below.

operand

Is an operand. 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 Chapter 3, *Expressions Reference*.

comparison_op

Is any of the following comparison operators:

- **Numeric.** EQ, IS, NE, IS_NOT, GE, GT, EXCEEDS, LT, LE.
- **Character.** EQ, IS, NE, IS_NOT, CONTAINS, OMMITS, IN (*list*).

The character operators IS and EQ enable you to 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. For example:

```
WHERE ZipCode IS '112$$'
```

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.

;

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 Chapter 1, *Language Rules Reference*.

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 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 needs to 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
SEGNAME=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=RDAT, 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 shown below in **bold**:

```
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 10 instances of the EmplInfo segment and places them into Stackemp:

```
FOR 10 NEXT Emp_ID INTO Stackemp;
```

Using Selection Logic to Retrieve Rows

The following example retrieves every instance of the EmpInfo segment that has a department value of MIS:

```
FOR ALL NEXT Emp_ID 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:

```
FOR ALL NEXT Emp_ID 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.

NEXT After a MATCH

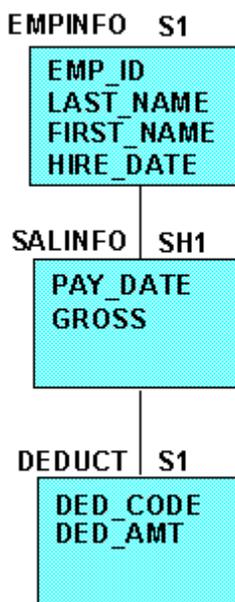
NEXT can also be used in conjunction with the MATCH command. This example issues a MATCH for employee ID. If there is not a match, a message is displayed. If there is a match, all the instances of the child segment for that employee are retrieved and placed in the stack Stackemp. The NEXT command can be coded as part of an ON MATCH condition, but it is not required because 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
OR ALL NEXT Dat_Inc INTO Stackemp;
```

Data Source Navigation Using NEXT: 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 EmplInfo segment which is the parent of the SalInfo segment based on its current position. The position for the EmplInfo segment can be established by either a prior MATCH or NEXT command. If no position has been established for the EmplInfo segment, an error occurs.

You can use the NEXT command:

- With one segment.
- With multiple segments.
- Following another NEXT or MATCH command.

Data Source Navigation: 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 3 instances of the EmplInfo 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:

```
FOR 3 NEXT Emp_ID INTO Stemp WHERE Department EQ 'MIS';
```

If there may not have been as many rows to be retrieved as you specified in the FOR prefix, you can determine how many rows were actually retrieved by checking the target stack's FocCount variable.

Data Source Navigation: 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 EmplInfo to the beginning of the data source (EmplInfo 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 EmplInfo 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.

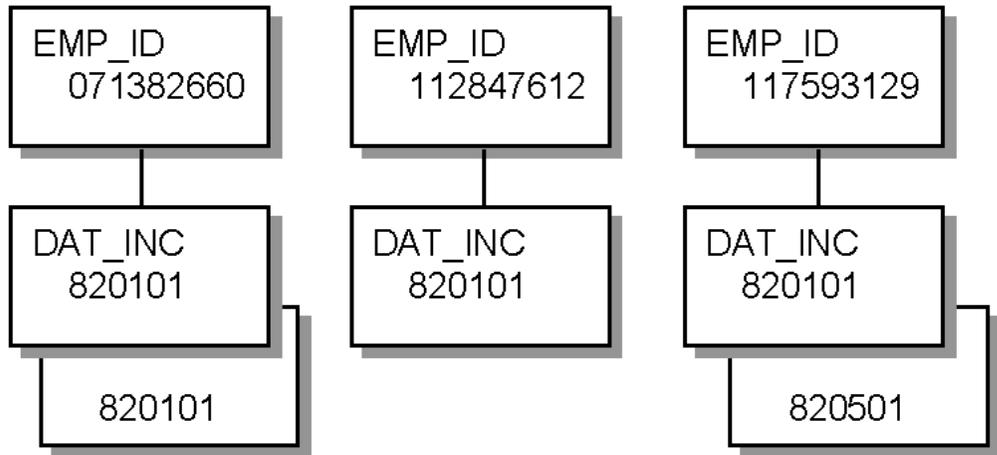
```
REPOSITION Emp_ID;
NEXT Emp_ID Pay_Date;
NEXT Emp_ID Pay_Date;
```

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, which is described in *Data Source Navigation: NEXT Following NEXT or MATCH* on page 2-75. 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 EmplInfo and SallInfo segments and places the output into the Stemp stack. The five records may or may not all have the same EmplInfo segment.

```
FOR 5 NEXT Emp_ID Dat_Inc INTO Stemp;
```

NEXT

If the data source is populated as follows



all of the fields from the following segment instances are added to the stack:

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

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: NEXT Following NEXT or MATCH

In order to 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 EmplInfo 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 EmplInfo segment and the next three instances of the SallInfo 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

NEXT

Every NEXT command that uses a FOR prefix does so independently of any other NEXT command. If there are two NEXT commands, the first executes and when it completes, 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 EmplInfo segment and places the instances into the stack. The second NEXT retrieves the next three instances of the SallInfo segment. Note its parent instance is the second EmplInfo segment instance. The stack variable FocCount indicates the number of rows currently in the stack. The prefix Stemp is needed to indicate which 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 looks like the following. 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 child segment's Gross field is greater than 1,000. Like the previous example, the data retrieved is only for the employee that the first NEXT retrieves:

```
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 retrieves the next 3 records where the child segment's Gross field is greater than 1,000. Like above, the data retrieved is only for the employee that the first NEXT retrieves:

```
NEXT Emp_ID;
FOR 3 NEXT Pay_Date INTO Stemp WHERE Gross GT 1000;
```

Unique Segments

One of the ways Maintain allows separate segments to be joined together is uniquely, that is, in a one to one relation. 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 command's field list. You cannot specify the unique segment as an anchor segment.

If an attempt is made to retrieve data from a unique segment and it 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 to the unique. 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—that is, 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.

The ON MATCH command defines the action to take if the prior MATCH command succeeds—that is, 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.

It is recommended that you 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 2-44.

Syntax

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;
```

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;
  PERFORM Jobs;
ENDBEGIN
```

ON NEXT

The ON NEXT command defines the action to take if the prior NEXT command succeeds—that is, 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 2-44.

Syntax **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 5 percent increase for the next employee in the data source:

```
NEXT Emp_ID;
ON NEXT COMPUTE NewSal = 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 = Curr_Sal * 1.05;
ENDBEGIN
```

ON NOMATCH

The ON NOMATCH command defines the action to take if the prior MATCH command fails—that is, 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 2-44.

Syntax

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—that is, 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 2-44.

Syntax

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 PERFORM Command

The syntax of the PERFORM command is

```
PERFORM functionname [;]
```

where:

functionname

Specifies the name of the function to perform.

;

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 Chapter 1, *Language Rules Reference*.

For example, to perform the function named NextSet, you would issue the command:

```
PERFORM NextSet;
```

Reference Commands Related to PERFORM

- **CASE/ENDCASE** defines a Maintain function. See *CASE* on page 2-14.
- **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();
```

For more information about terminating commands with semicolons, see *Terminating a Command's Syntax* in Chapter 1, *Language Rules Reference*.

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 `Nether` is performed after a `NOMATCH` condition occurs:

```
ON NOMATCH PERFORM Nether
```

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 and PERFORM* on page 2-49 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.

Syntax **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 execute. 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, it is recommended that the expression 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 Chapter 3, *Expressions Reference*

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 counter's initial value 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 counter's value 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 counter's format. It can be any valid Maintain format except for TX. The format is required only if you are defining the variable in this command.

init_expr

Is an expression whose value is assigned to the counter before the first iteration of the loop. It can be any valid Maintain expression.

increment_expr

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 Chapter 1, *Language Rules Reference*.

REPEAT

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 execute 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:

```
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:

```
REPEAT 27;
```

You can also specify a condition that must be true or false for looping to continue:

```
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.

```
FOR ALL NEXT Country INTO Stk1;
COMPUTE Cnt = 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 = 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 = 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 Stackemp, so each NEXT puts data into the same row of the stack. In other words, INTO Stackemp is the same as INTO Stackemp(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

```
REPEAT 10;  
    NEXT Emp_ID INTO Stackemp;  
    .  
    .  
    .  
ENDREPEAT
```

is the same as:

```
REPEAT 10 Cnt=1;  
    NEXT Emp_ID INTO Stackemp;  
    .  
    .  
    .  
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=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 Stackemp 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=1;  
    COMPUTE Stkemp(Cnt).NewSal=Stkemp(Cnt).Curr_Sal * 1.05;  
    IF Cnt GE Stackemp.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 execute 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 execute 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 = 1;
  TYPE "In A loop with A = <A>";
  REPEAT UNTIL B GT 4; B = 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. For information about the COMPUTE command, see *COMPUTE* on page 2-20.
- **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:

```
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 that segment's chain. 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, it is recommended that you ensure that you begin searching from the beginning of the data source or path by first issuing the REPOSITION command.

Syntax **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, 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 Chapter 1, *Language Rules Reference*.

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. See *NEXT* on page 2-65.
- **MATCH** searches entire segments for a matching field value and can retrieve an exact match in the data source. See *MATCH* on page 2-62.

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 segment's instances in the stack against the corresponding instances in the data source. If an instance's keys 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 anchor's ancestor segments, or provide ancestor segment key values in the source stack. This enables REVISE to navigate from the root to the anchor segment's first instance.

Syntax

REVISE Command

The syntax of the REVISE command is

```
[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 stack's rows 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 via 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.

;

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 Chapter 1, *Language Rules Reference*.

Example Using REVISE

In the following example the user is able to enter information for a new employee, or change an existing employee's last name. All of the information is stored in a stack named EmpStk.

```

MAINTAIN FILE EMPLOYEE
FOR ALL NEXT Emp_ID INTO EmpStk;
WINFORM SHOW GetData;

CASE Alter_Data
FOR ALL REVISE Last_Name FROM EmpStk;
ENDCASE
END

```

ROLLBACK

When function `Alter_Data` is called from a form's event handler, the `REVISE` command reads `EmpStk` and tries to find each row's `Emp_ID` in the `Employee` data source. If `Emp_ID` exists in the data source, `REVISE` updates that segment instance's `Last_Name` field. If it does not exist, then `REVISE` inserts a new `EmplInfo` instance into the data source, and writes `EmplInfo`'s fields from the stack to the new instance.

Reference Usage Notes for REVISE

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

Reference Commands Related to REVISE

- **INCLUDE** adds new segment instances to a data source. See *INCLUDE* on page 2-53.
- **UPDATE** updates data source fields. See *UPDATE* on page 2-109.
- **COMMIT** makes all data source changes since the last `COMMIT` permanent. See *COMMIT* on page 2-18.
- **ROLLBACK** cancels all data source changes made since the last `COMMIT`. See *ROLLBACK* on page 2-94.

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 via the `COMMIT` command.

Syntax ROLLBACK Command

The syntax of the `ROLLBACK` command is

```
ROLLBACK [;]
```

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 Chapter 1, *Language Rules Reference*.

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.

```

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.

RUN

You can execute a compiled procedure by using the RUN command.

This command is outside the Maintain language, but is described here in *Command Reference* for your convenience. You can issue this command from within an external procedure, not from within a Maintain procedure.

Syntax **RUN Command**

The syntax of the RUN command is

```
RUN procedure_name
```

where:

```
procedure_name
```

Is the name of the compiled procedure.

Example **Using Run**

For example, the following command executes the compiled version of the EmpInfo procedure:

```
RUN EmpInfo
```

You can create a compiled procedure using the COMPILE command.

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 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 SAY command.

Syntax **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 message's last line.

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 FILEDEF command, or a DYNAM command on OS/390 or MVS) 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 FILEDEF or 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. In a:

- Web-deployed application, the message is written to the Web page currently displayed in the Web browser.

If the message is written from a procedure that resides on a different server than the Web page, the message is prefixed with "(FOC03764) From Server ==> " to indicate that it was posted by a remote procedure.

STACK CLEAR

- Windows-deployed application run from the Maintain Development Environment, the message is written to the Output window's Run tab.

In addition, if TO *ddname* is omitted and this procedure was called remotely (that is, called via a CALL *procname* AT command), the message will also be copied to the calling procedure's FocMsg stack.

Reference Commands Related to SAY

TYPE writes messages to a file or to a Windows form. See *TYPE* on page 2-105.

Writing Segment and Stack Values

You can use the SEG and STACK prefixes to write the values of all a segment's fields or a stack's columns 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 specified segment's fields.

STACK.*stackname(row)* inserts, for the specified stack, the specified row's values.

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 only 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.

STACK CLEAR

STACK CLEAR clears the contents of each of the stacks listed, so that each stack has no rows. This sets each stack's FocCount variable to zero and FocIndex variable to one.

Syntax 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, 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 Chapter 1, *Language Rules Reference*.

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 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 Chapter 1, *Language Rules Reference*.

Example Using STACK SORT

The following sorts the stack Emp using the values in the stack column Emp_ID:

```
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:

```
STACK SORT Emp BY HIGHEST Salary;
```

The next example sorts the stack by Department. Within Department the rows are ordered by highest Salary:

```
STACK SORT Emp BY Department BY HIGHEST Salary;
```

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, and assists in managing specific WebFOCUS server environment settings.

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.
- Retrieve DBMS error codes, allowing developers to code applications to efficiently recover from error conditions.
- Issue native SQL commands directly from a Maintain procedure.

For use with WebFOCUS Servers, the SYS_MGR syntax can be used from within a Maintain procedure to:

- Issue system settings for WebFOCUS servers on the fly.
- Set values for Userid and Password for target servers before issuing an EXEC AT or CALL AT.

Deactivating Preliminary Database Operation Checking With SYS_MGR.PREMATCH

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;
```

and then 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.PREMATCH. 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 PREMATCH 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

Setting PREMATCH

The syntax is

```
SYS_MGR.SET_PREMATCH{0|1};
```

or

```
SYS_MGR.PREMATCH = {0|1}
```

where:

0

Disables prematching.

1

Turns on prematching

To check the current setting for pre-match, use:

```
SYS_MGR.GET_PREMATCH();
```

or

```
SYS_MGR.PREMATCH;
```

Example Setting PREMATCH Off

Suppose you have a Maintain procedure with the following code:

```
SYS_MGR.PREMATCH = 0; /* stop pre-selecting */
FOR ALL INCLUDE PRODUCTS FROM PRODSTACK;
SYS_MGR.PREMATCH = 1; /* restore */
```

If PRODSTACK had 5000 rows, setting PREMATCH to 0 before the INCLUDE reduced the number of database engine interactions from 10,000 to 5,000.

Retrieving DBMS Codes With 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 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;
```

Issuing Native SQL Commands (SQL Passthru):

You can issue DBMS commands directly 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 **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 your iWay documentation.

command

Is any valid SQL command, including CREATE, DROP, and INSERT.

Example **Issuing 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 **Passing Credentials to Remote Servers With SYS_MGR.ENGINE**

You can easily set user and password values for remote WebFOCUS Servers using the SYS_MGR.ENGINE command, and setting *enginename* to EDA. If you do this, you do not need to use EDASPROF for passing the SQL EDA SET USER command.

The following code takes values for userid and password from the launch form and passes them to the target server:

```
COMPUTE USER/A8;
      PWD/A8;
iwc.findappcgivalue("IBIC_user",USER);
iwc.findappcgivalue("IBIC_pass",PWD);
.
.
.
SYS_MGR.ENGINE("EDA",'SET USER EDASERVE/' || USER || ',' || PWD);
EXEC myreport AT EDASERVE into mystack
```

In the case of a remote server requiring different userid and password values than those supplied at login time, you can omit the use of the `iwf.findappcvalue` function and instead design the application to gather values from the end user at runtime.

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 mntbtabs
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
```

Setting System Values for a WebFOCUS Server With SYS_MGR.FOCSET

Using the `SYS_MGR.FOCSET`, you can set certain environment settings for the WebFOCUS Server.

Syntax SYS_MGR.FOCSET Command

The syntax is

```
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
```

value

Is an appropriate setting for that command.

Example Setting DEFCEM From a Maintain Procedure

The following code

```
MAINTAIN
COMPUTE MYDATE/YYMD;
SYS_MGR.FOCSET("DEFCEM", "21");
COMPUTE DATA1/YMD='90/01/01';
COMPUTE MYDATE=DATE1;
TYPE "After setting DEFCEM=21, MYDATE=<<MYDATE";
END
```

produces the following output:

```
After setting DEFCEM=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's session.

TYPE

The TYPE command writes messages to a file, a Web browser, or the Maintain Development Environment's Output window. 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 **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 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 FILEDEF or DYNAM command.

If ON *ddname* is omitted, in a:

- Web-deployed application, the message is written to the Web page currently displayed in the Web browser.
If the message is written from a procedure that resides on a different server than the Web page, 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 Output window's Run tab.

In addition, if ON *ddname* is omitted and this procedure was called remotely (that is, called via a CALL *procname* AT command), the message will also be copied to the calling procedure's FocMsg stack.

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, 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 Chapter 1, *Language Rules Reference*.

Reference **Commands Related to TYPE**

SAY writes messages to a file or to the server console; messages can include multiple expressions of all types. See *SAY* on page 2-97.

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 in column 20";
TYPE "Skip <+8 8 spaces within text";
TYPE "Back up <-4 4 spaces and overwrite";
```

results in:

```
This starts in column 20
Skip      8 spaces within text
Back4 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.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:

```
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:

```
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:

```
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 anchor's ancestor segments, or provide ancestor segment key values in the source stack or Current Area. This enables UPDATE to navigate from the root to the anchor segment's first instance.

Syntax

UPDATE Command

The syntax of the UPDATE command is

```
[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.

UPDATE

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, 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 Chapter 1, *Language Rules Reference*.

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 are changed without first finding out if they already exist 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.
- After an UPDATE command completes, 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 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. See *COMMIT* on page 2-18.
- **ROLLBACK** cancels all data sources changes made since the last COMMIT. See *ROLLBACK* on page 2-94.

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 is allowed to enter 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 is performed which tries to clean up the data. The IF command which starts at the beginning of the function checks to see if there are any rows in the stack. If the stack does not have any rows, then a form is displayed allowing the user to enter new data. If the stack has rows, it is because the user made a mistake, so a different form is displayed allowing the user to edit the entered data.

The Maintain procedure contains:

```

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 data source's position 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 an application's user interface, whereas a procedure controls the application's logic and use of data.

Syntax **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 form's controls, 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 of the form's 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 of the form's controls, dynamically at run time. SHOW_INACTIVE does not display the form.

HIDE

In Windows-deployed applications, this removes the specified form from the screen. You can hide any form except for the active one. You can later redisplay it using the WINFORM UNHIDE command.

In Web-deployed applications, this option does not apply.

UNHIDE

In Windows-deployed applications, this is used to display a form which has been hidden using the WINFORM HIDE command, or which has been covered on the screen by other forms. If multiple forms are on the screen, the specified form is displayed on top. This command does not make the specified form active. If the specified form was already displayed and unobstructed by other forms, this command is ignored.

As an alternative, the application end user can unhide a partially covered form by clicking it with the mouse.

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 form's data values 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 on the Web, this avoids returning the end user to the launch form. Any client-level logic—such as hypertext 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 the color of an entire form, or if you are getting the name of the control that has input focus; otherwise you must specify it.

property

Is any valid property. For information about properties, see Chapter 4, *Form and Control Properties Reference*.

value

Is a value that is valid for the specified property. Properties and their values are described in Chapter 4, *Form and Control Properties Reference*.

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. For information about properties and their values, see Chapter 4, *Form and Control Properties Reference*.

;

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 Chapter 1, *Language Rules Reference*.

For information about using the WINFORM SET and GET commands, see *Dynamically Changing Form Control Properties* on page 2-116.

Reference **Commands Related to WINFORM**

- **NEXT** retrieves sets of data from a data source into a stack; you can then display the stack's data in a form, as described in *NEXT* on page 2-65.
- **TYPE** displays messages on the screen or writes them to a file. See *TYPE* on page 2-105.

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. A variable's default is determined by its data type and whether it was defined with the MISSING attribute:

Data Type	Default value without the MISSING attribute	Default value with the MISSING attribute
Character	space	null
Numeric	zero	null
Date and time	space	null

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 a form's properties 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, WINFROM SHOW will make the form active; in Web-deployed applications, it will display the form). If you wish to change a form's properties in response to user activity in the form, you can trigger a function containing WINFORM SET commands from those user events. You cannot dynamically set a form's properties before it has been opened with either a WINFORM SHOW or WINFORM SHOW_INACTIVE.

For example, you could develop a data entry function that checks if a user has entered data into a field; if the user has not, you could use the WINFORM SET command to change the field's color and give it focus, effectively drawing the user's attention to it and making it the target of any keyboard activity.

For information about form and control properties, see Chapter 4, *Form and Control Properties Reference*.

CHAPTER 3

Expressions Reference

Topics:

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

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.

For details on Maintain built-in functions that you can use in expressions see the *WebFOCUS Using Functions* manual.

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:

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

A numeric expression returns a numeric value. For details see *Writing Numeric Expressions* on page 3-3.

- **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:

```
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 and Time Expressions* on page 3-8.
- Date-time expressions.
- **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:

```
COMPUTE First_Init/A1 = MASK (First_Name, '9$$$$$$$$') ;
```

A character expression returns a character value. For details see *Writing Character Expressions* on page 3-37.

- **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 3-41.
- **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 3-44.

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 returns a number.

A numeric expression can consist of the following components, shown below in **bold**:

- 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 = CURR_SAL * 0.05 ;
```

- A numeric function. For example:

```
COMPUTE LONGEST_SIDE = MAX (WIDTH, HEIGHT) ;
```

- Two or more numeric expressions joined by a numeric operator. For example:

```
COMPUTE PROFIT = (RETAIL_PRICE - UNIT_COST) * UNIT_SOLD ;
```

Reference **Numeric Operators**

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

Addition	+
Subtraction	-
Multiplication	*
Division	/
Integer division	DIV
Remainder division	MOD
Exponentiation	**

Note: 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 **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 **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 DbIMod 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;
           DbIMod/D4.1=StkCar (Cnt) .MPG MOD 10;
    TYPE "MPG=<<StkCar (Cnt) .MPG "
        "IntMod=<<IntMod   DbIMod=<<DbIMod"
ENDREPEAT Cnt=Cnt+1;
END

```

The decimal place in the variable DbIMod is truncated, even though the format is D4.1.

```

MPG=      16 INTMOD=    6 DBLMOD=  6.0
MPG=       9 INTMOD=    9 DBLMOD=  9.0
MPG=      11 INTMOD=    1 DBLMOD=  1.0
MPG=      25 INTMOD=    5 DBLMOD=  5.0

```

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:

```

COMPUTE PROFIT = RETAIL_PRICE - UNIT_COST * UNIT_SOLD ;
COMPUTE PROFIT = (RETAIL_PRICE - UNIT_COST) * UNIT_SOLD ;

```

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.

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

Example Identical Operand Formats

Because the following variables are defined as integers,

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

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

```
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.

Order	Format
1	16-byte packed decimal
2	Double-precision floating point
3	8-byte packed-decimal
4	Single-precision floating point
5	Integer
6	Character (alphanumeric and text)

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.

A character (that is, alphanumeric or text) value can be used in a computation if it is a numeric string. Maintain attempts to convert the character operand to the format of the other operand in the expression. If both operands are character, Maintain tries to convert them to double-precision. If the conversion is not possible, Maintain generates an error.

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

Continental Decimal Notation

When the Continental Decimal Notation feature is in effect (that is, when CDN has been set to ON), if you specify a decimal constant in a command using a comma to indicate the decimal position, you need to delimit the entire value using single or double quotation marks. This is to be done in Maintain commands only, not in data entered in forms at run time. For details about the SET CDN command see your WebFOCUS Developer Studio documentation.

Writing Date and Time 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, shown below in **bold**:

- 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;
```

For information about working with cross-century dates, see your WebFOCUS Developer Studio documentation.

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:

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

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

Value	Date Format For example: MDY		Integer, Packed, or Alphanumeric Format For example: A6MDY	
	Stored	Displayed	Stored	Displayed
October 31, 1992	33542	10/31/92	103192	10/31/92
November 01, 1992	33543	11/01/92	110192	11/01/92

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:

```
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:

	Value in Date Format	Value in Integer Format
ShipDate = November 1, 1992	33543	110192
OrderDate = October 31, 1992	33542	103192
TurnAround	1	7000

If the date variables are in integer format, you can convert them to date format and then calculate TurnAround:

```
COMPUTE NewShipDate/MDY = ShipDate;  
NewOrderDate/MDY = OrderDate;  
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.

```
COMPUTE TURNAROUND/I4 = SHIPDATE - ORDERDATE ;  
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:

```
COMPUTE STARTDATE/MDY = '022893' ;
```

The following example calculates the number of days elapsed since January 1, 1993:

```
COMPUTE YEARTODATE/I4 = CURR_DATE - 'JAN 1 1993' ;
```

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:

```
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

```
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: 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:

```
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, which are 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, 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:

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

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 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.

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.

- 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

```
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:

```
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:

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

Specifying Date-Time Values

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.
- Via a WebFOCUS Maintain application form.

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.

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 a report procedure's IF expression, if the value contains no blanks or special characters, the single quotation marks are not necessary)

```
'date_string [time_string]'
```

```
'time_string [date_string]'
```

where:

time_string

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

```
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
```

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.

date_string

Can have one of the following three formats:

- The **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 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 *Hnn* format described in *Date-Time Display Formats* on page 3-17. The following are examples of numeric string date constants:

```
99
1999
19990201
```

- The **formatted-string format** contains a one or two-digit day, a one or two-digit month, and a two 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 DATEFORMAT setting. The following are examples of formatted-string date constants:

1999/05/20
5 20 1999
99.05.20
1999-05-20

- The **translated-string format** contains the full or abbreviated month name. The year must also be present in four-digit or two-digit form. If the day is missing, day 1 of the month is assumed; if present, it can have one or two digits. If the string contains both a two-digit year and a two-digit day, they must be in the order given by the DATEFORMAT setting. For example:

January 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.

Describing Date-Time Values

In a Master File, the USAGE format for a date-time field describes which components to display and various options for displaying them. In Master Files for data sources other than FOCUS, date-time fields must also have an ACTUAL format that indicates how the date-time value is stored in that data source.

The MISSING attribute for date-time fields can be ON or OFF. If it is OFF, and the date-time field has no value, it defaults to blank.

This section discusses:

- USAGE formats for displaying date-time field values.
- Alphanumeric formats for date-time values entered by a user at a terminal, read from a transaction data source, or embedded in an expression.
- ACTUAL formats for date-time fields.

Date-Time Display Formats

The USAGE (or FORMAT) attribute determines how date-time field values are displayed in report output and forms, and how they behave in expressions and functions; for FOCUS data sources, it also determines how they are stored. A new format type, H, describes date-time fields. The USAGE attribute for a date-time field contains the H format code and can identify either the length of the field or the relevant date-time display options.

The USAGE attribute can be one of the following

`USAGE = Hnn`

`USAGE = Htimefmt1`

`USAGE = Hdatefmt [separator] [timefmt2]`

where:

`Hnn`

Is the USAGE value for a numeric date-time value without date-time display options. This format is appropriate for use in alphanumeric HOLD data sources or transaction data sources.

`nn` is the field length, from 1 to 20, including up to eight characters for displaying the date and up to nine or 12 characters for the time. For lengths less than 20, the date is truncated on the right.

An eight-character date includes four digits for the year, two digits for the month, and two digits for the day of the month, YYYYMMDD.

A nine-character time includes two digits for the hour, two digits for the minute, two digits for the second, and three digits for the millisecond, HHMMSSsss. The millisecond component represents the decimal portion of the second to three places.

A twelve-character time includes two digits for the hour, two digits for the minute, two digits for the second, three digits for the millisecond, and three digits for the microsecond, HHMMSSsssmmm. The millisecond component represents the decimal portion of the second value to three places. The microsecond component represents three additional decimal places beyond the millisecond value.

With this format, there are no spaces between the date and time components, no decimal points, and no spaces or separator characters within either component. The time must be entered using the 24-hour system. For example, the value

`19991231225725333444` represents `1999/12/31 10:57:25.333444PM`

Htimefmt1

Is the USAGE format for displaying time only. Hour, minute, and second components are always displayed separated by colons (:), with no intervening blanks.

Unless you specify one of the AM/PM time display options, the time component is displayed using the 24-hour system.

When the format includes more than one time display option:

- The options must appear in the order hour, minute, second, millisecond, microsecond.
- The first option must be either hour, minute, or second.
- No intermediate component can be skipped. That is, if hour is specified the next option must be minute, it cannot be second.

The following table lists the valid time display options for a time-only USAGE attribute. Assume the time value is 2:05:27.123456 a.m.

Option	Meaning	Effect
H	hour (two digits) If the format includes the option <i>a</i> or <i>A</i> , the hour value is from 01 to 12. Otherwise, the hour value is from 00 to 23, with 00 representing midnight.	Prints a two-digit hour. For example: <code>USAGE = HH</code> prints <code>02</code>
h	hour with zero suppression If the format includes the option <i>a</i> or <i>A</i> , the hour value is from 1 to 12. Otherwise, the hour is from 0 to 23.	Displays the hour with zero suppression. For example: <code>USAGE = Hh</code> prints <code>2</code>
I	minute (two digits) The minute value is from 00 to 59.	Prints the two-digit minute. For example: <code>USAGE = HHI</code> prints <code>02:05</code>

Option	Meaning	Effect
i	minute with zero suppression The minute value is from 0 to 59.	Prints the minute with zero suppression. Cannot be used together with an hour format (H or h). For example: <code>USAGE = Hi</code> prints 5
S	Second (two digits) 00 to 59	Prints the two-digit second. For example: <code>USAGE = HHIS</code> prints 02:05:27
s	millisecond (three digits — after the decimal point in the second) 000 to 999	Prints the second to three decimal places. For example: <code>USAGE = HHISs</code> prints 02:05:27.123
m	microsecond (three additional digits after milliseconds) 000 through 999	Prints the second to six decimal places. For example: <code>USAGE = HSsm</code> prints 27.123456
A	12-hour time display with AM or PM in uppercase.	Prints hours from 01 to 12 followed by AM or PM. For example: <code>USAGE = HHISA</code> prints 02:05:27AM
a	12-hour time display with am or pm in lowercase.	Prints hours from 01 to 12 followed by am or pm. For example: <code>USAGE = HHISa</code> prints 02:05:27am

Hdatefmt

Is the USAGE format for displaying the date portion of the date-time field.

The date components can be in any of the following combinations and order:

- Year first combinations: Y, YY, YM, YYM, YMD, YYMD
- Month-first combinations: M, MD, MY, MYY, MDY, MDYY
- Day-first combinations: D, DM, DMY, DMYY

The date format can include the following display options as long as they conform to the allowed combinations. In the following table, assume the date is February 5, 1999.

Option	Meaning	Example
Y	2-digit year	99
YY	4-digit year	1999
M	2-digit month (01 - 12)	02
MT	Full month name	February
Mt	Short month name	Feb
D	2-digit day	05
d	zero-suppressed day	5
k	For formats in which month or day is followed by year and month is translated to a short or full name, separates the year from the day with a comma and blank. Otherwise the separator is a blank.	USAGE = HMtDkYY prints Feb 05, 1999

separator

Is a separator between the date components. The default separator is a slash (/). Other valid separators are: period (.), hyphen (-), blank (B), or none (N). With translated months, these separators can only be specified when the k option is not used.

timefmt2

Is the format for a time that follows a date. Time is separated from the date by a blank; time components are separated from each other by colons. Unlike the format for time alone, a time format that follows a date format consists of at most two characters: a single character to represent all of the time components to be displayed and, optionally, one character for an AM/PM option. The following table lists the valid options. Assume the date is February 5, 1999 and the time is 02:05:25.444555 a.m.

Option	Meaning	Example
H	Prints hours	USAGE = HYYMDH prints 1999/02/05 02
I	Prints hours:minutes	USAGE = HYYMDI prints 1999/02/05 02:05
S	Prints hours:minutes:seconds	USAGE = HYYMDS prints 1999/02/05 02:05:25
s	Prints hours:minutes:seconds.milliseconds	USAGE = HYYMDS prints 1999/02/05 02:05:25.444
m	Prints hours:minutes:seconds.microseconds	USAGE = HYYMDm prints 1999/02/05 02:05:25.444555
A	Prints AM or PM. Uses the 12-hour system and causes the hour to be printed with zero suppression.	USAGE = HYYMDSA prints 1999/02/05 2:05:25AM
a	Prints am or pm. Uses the 12-hour system and causes the hour to be printed with zero suppression.	USAGE = HYYMDSa prints 1999/02/05 2:05:25am

Note: Unless you specify one of the AM/PM time display options, the time component is displayed using the 24-hour system.

Example Reading Date-Time Values From a Transaction Data Source

The DTTRANS fixed-format transaction data source has a two-digit ID field and a date-time field that contains both the date (as eight characters) and time (in the format hour:minute:second):

```
0120000101 02:57:25
0219991231 14:05:35
```

Because the transaction data source contains the dates in numeric string format the DATEFORMAT setting is not used, and the dates are entered in YMD order.

The following transaction data source is also valid:

```
0101/01/2000 02:57:25
0212/31/1999 14:05:35
```

The following Master File describes the FOCUS data source named DATETIME, which will receive these values:

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

The following Maintain procedure loads the transaction values into the FOCUS data source:

```
MAINTAIN FILE DTTrans AND DateTime
FOR ALL NEXT ID DT1 INTO DateTimeStack;
FOR ALL INCLUDE DateTime.ID FROM DateTimeStack;
END
```

To see the printed values, call the following external report procedure

```
TABLE FILE DATETIME
PRINT ID DT1
END
```

which generates this report:

```
ID  DT1
--  ---
 1  2000/01/01 02:57:25
 2  1999/12/31 14:05:35
```

Example Using a Date-Time Value in a COMPUTE Command

```
COMPUTE RAISETIME/HYYMDIA = DT(20000101 09:00AM);
```

ACTUAL Formats for Date-Time Values

ACTUAL formats supported for date-time values are:

- *Ann*, H8, H10, and H12. *Ann* accepts all the date-time string formats described in *Specifying Date-Time Values* on page 3-14, as well as the *Hnn* USAGE display format described in *Date-Time Display Formats* on page 3-17. ACTUAL=H8, H10, or H12 accepts a date-time field as it occurs in a binary HOLD data source or SAVB data source. ACTUAL=*Ann* accepts a date-time field as it occurs in an alphanumeric HOLD data source or SAVE data source.
- RDBMS-specific H formats. See *ACTUAL Format Conversion Chart for Relational Data Sources* on page 3-23 for additional information about ACTUAL formats used with relational Data Adapters. The corresponding USAGE must be an H format.

Reference ACTUAL Format Conversion Chart for Relational Data Sources

The following table describes date-time support for RDBMS data types:

RDBMS	RDBMS DATA TYPE	ACTUAL FORMAT
DB2, SQL/DS	TIME	HHIS
	TIMESTAMP	HYYMDm
	DATE	DATE
Oracle	DATE	HYYMDS
Sybase, MSS	DATETIME (called TIMESTAMP internally)	HYYMDs
	SMALLDATETIME	HYYMDI
Teradata	DATE	DATE
Informix	DATETIME	HYYMDH HHIS HYYMDI HHISs HYYMDS HHI HYYMDS HH
	DATE	DATE
ODBC	TIME(<i>scale</i>) where <i>scale</i> indicates that the time includes a fraction of the second.	FORMAT Scale HHIS 0 HHISs 3 HHISm 6
	TIMESTAMP(<i>precision, scale</i>) where <i>precision</i> indicates the print length and <i>scale</i> indicates the number of decimal places for fraction of the second.	FORMAT Precision, scale HYYMD 10, 0 HYYMDH 13, 0 HYYMDI 16, 0 HYYMDS 19, 0 HYYMDS 23, 3 HYYMDm 26, 3

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 *Manipulating Date-Time Values With Subroutines* on page 3-27.

Comparison and Assignment

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:

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

Field DT1 supports missing values. Consider the following request:

```
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:

```
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
HCVRT	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.
HDATE	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.

Note:

- In those arguments that give you a choice of 8 or 10, use 8 for processing values without microseconds, 10 when the field value includes microseconds.
- In a Maintain procedure, the last argument is always the name of the variable to which you are assigning the function's return value. In a report procedure, the last argument is always a USAGE format that indicates the data type returned by the function; the type may be A (alpha), I (integer), D (double precision), DATE (date), or H (date-time).

All of the examples in the following topics use the DATETIME data source created in *Reading Date-Time Values From a Transaction Data Source* on page 3-21.

Reference Component Names and Values for Use With Date-Time Functions

The following component names and values are supported as arguments to those date-time functions that require you to specify a component name as an argument:

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

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. In this implementation, you can control the date format used to enter date-time values with the DATEFORMAT parameter. For details, see *Specifying Date-Time Values* on page 3-14.
- The international standard for the first day of a week is Monday. You can use the WEEKFIRST parameter to control the day used as the first day of the week by the date-time functions.
- 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.

Manipulating Date-Time Values With Subroutines

The full set of date-time subroutines that is available for reporting is also available for data maintenance applications. However, the required syntax and implementation is slightly different. In a Maintain procedure, the last argument of a date-time function is always the name of the variable to which you are assigning the function's return value.

For a list of date-time subroutines see *Date-Time Subroutines* on page 3-25.

All of the examples in the following topics use the DATETIME data source.

Example **Converting a Date-Time Field to Alphanumeric Format**

The following request converts the DT1 field to alphanumeric format: .

```
MAINTAIN FILE DATETIME
FOR ALL NEXT ID INTO STK;
COMPUTE
RESULT_HCNVRT/A20 = HCNVRT(STK.DT1, '(HYYMDH)', 20, RESULT_HCNVRT);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "RESULT_HCNVRT = " RESULT_HCNVRT;
END
```

Syntax **How to Convert an Alphanumeric String to a Date-Time Value**

You can use the HINOUT subroutine in an expression to convert an alphanumeric string to a date-time value. The syntax is: .

```
HINPUT (inputlength, 'inputstring', {8|10}, output)
```

where:

inputlength

Is the length of the alphanumeric string to convert. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value.

inputstring

Is the alphanumeric string to convert. You can supply the actual string enclosed in single quotation marks, the name of an alphanumeric field, or an expression that returns an alphanumeric value. The alphanumeric string can consist of any valid date-time input value as described in *Specifying Date-Time Values* on page 3-14.

8|10

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for time values down to microseconds.

output

Is the returned date-time value.

Example **Converting an Alphanumeric String to a Date-Time Value**

The following request converts the DT1 field to alphanumeric format:

```
MAINTAIN FILE DATETIME
COMPUTE
RESULT/HMtDYmA = HINPUT(20, '19971029133059888999', 10, RESULT);
TYPE RESULT;
END
```

Reference **Component Names and Values for Use With Date-Time Functions**

The following component names and values are supported as arguments to those date-time functions that require you to specify a component name as an argument:

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

Syntax **How to Increment a Date-Time Field**

You can use the new HADD subroutine in an expression to increment a date-time field by a given number of units: for example, 1 year, 3 months, or -15 seconds. The syntax is:

```
HADD (dtfield, 'component', increment, {8|10}, output)
```

where:

dtfield

Is the date-time value to increment. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

component

Is the name of the component to be incremented, enclosed in single quotation marks. See *Component Names and Values for Use With Date-Time Functions* on page 3-29 for a list of supported components.

increment

Is the number of units by which to increment the specified component. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value.

8|10

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for time values down to microseconds.

output

Is the returned date-time value.

Example **Incrementing the Month Component of a Date-Time Field**

The following request adds two months to the DT1 field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID DT1 INTO DTSTK
COMPUTE
NEW_DATE/HYYMDS = HADD(DTSTK.DT1, 'MONTH', 2,10, NEW_DATE);
TYPE "DT1 IS:                <DTSTK(1).DT1  "
TYPE "NEW_DATE IS: <NEW_DATE  "
```

The output is:

```
DT1 IS:          2000/1/1 02:57:25
NEW_DATE IS:    2000/3/1 02:57:25

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

Note that the day is adjusted to be valid for the resulting month.

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

You can use the new HDIFF subroutine in an expression to find the number of boundaries of a given type crossed in going from date 2 to date 1. The syntax is:

```
HDIFF (dtfield1, dtfield2, 'component', output)
```

where:

dtfield1

Is the ending date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

dtfield2

Is the starting date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

component

Is the name of the component to be used in the calculation, enclosed in single quotation marks. If the unit is weeks, the WEEKFIRST setting is used in the calculation. See *Component Names and Values for Use With Date-Time Functions* on page 3-29 for a list of supported components.

output

Is the returned date-time value.

Example Finding the Number of Days Between Two Date-Time Fields

The following request finds the number of days between the ADD_MONTH and DT1 fields:

```

MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
NEW_DATE/HYYMDS = HADD(STK.DT1, 'MONTH', 2,10, NEW_DATE);
DIFF_DAYS/D12.2 = HDIFF(NEW_DATE,STK.DT1,'DAY', DIFF_DAYS);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "NEW_DATE = "NEW_DATE;
TYPE "DIFF_DAYS = "DIFF_DAYS
END

```

Syntax How to Extract a Date-Time Component in Alphanumeric Format

You can use the HNAME subroutine in an expression to extract a specified component from a date-time field and return it in alphanumeric format. The syntax is:

```
HNAME (dtfield, component, output)
```

where:

dtfield

Is the date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

component

Is the name of the component to be extracted, enclosed in single quotation marks. See *Component Names and Values for Use With Date-Time Functions* on page 3-29 for a list of supported components.

output

Is the alphanumeric returned component. All other components are converted to strings of digits only. The year is always four digits, and the hour assumes the 24-year system.

Example **Extracting the Day Component in Alphanumeric Format From a Date-Time Field**

The following request extracts the day in alphanumeric format from the DT1 field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DAY_COMPONENT/A2=HNAME(STK.DT1, 'DAY', DAY_COMPONENT);
TYPE "STK(1).DT1      = "STK(1).DT1";
TYPE "DAY_COMPONENT = <DAY_COMPONENT"
END
```

Syntax **How to Extract a Date-Time Component in Numeric Format**

You can use the new HPART subroutine in an expression to extract a specified component from a date-time field and return it in numeric format. The syntax is:

```
HPART (dtfield, 'component', output)
```

where:

dtfield

Is the date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

component

Is the name of the component to be extracted, enclosed in single quotation marks. See *Component Names and Values for Use With Date-Time Functions* on page 3-29 for a list of supported components.

output

Is the integer of the returned component. The year is always four digits, and the hour assumes the 24-hour system.

Example **Extracting the Day Component in Numeric Format From a Date-Time Field**

The following request extracts the day in integer format from the DT1 field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DAY_COMPONENT/I2 = HPART(STK.DT1, 'DAY', DAY_COMPONENT);
TYPE "STK(1).DT1      = <STK(1).DT1";
TYPE "DAY_COMPONENT = <DAY_COMPONENT";
END
```

Syntax **How to Insert a Component Into a Date-Time Field**

You can use the new HSETPT subroutine in an expression to insert the numeric value of a specified component into a date-time field. The syntax is:

```
HSETPT (dtfield, 'component', value, {8|10}, output)
```

where:

dtfield

Is the date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

component

Is the name of the component to be inserted, enclosed in single quotation marks. See *Component Names and Values for Use With Date-Time Functions* on page 3-29 for a list of supported components.

value

Is the numeric value to use for the requested component. You can supply the actual value, the name of a numeric field that contains the value, or an expression that returns the value.

8|10

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for time values down to microseconds.

output

Is the returned date-time value.

Example **Inserting the Day Component Into a Date-Time Field**

The following request inserts the day into the ADD_MONTH field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
  ADD_MONTH/HYYMDS = HADD(STK.DT1, 'MONTH', 2, 8, ADD_MONTH);
  INSERT_DAY/HYYMDS = HSETPT(ADD_MONTH, 'DAY', 28, 8, INSERT_DAY);
TYPE "STK(1).DT1" = <STK(1).DT1";
TYPE "ADD_MONTH" = <ADD_MONTH";
TYPE "INSERT_DAY" = <INSERT_DAY";
END
```

Syntax **How to Set the Time Portion of a Date-Time Field to Midnight**

You can use the new HMIDNT subroutine in an expression to change the time portion of a date-time field to midnight (all zeroes). This function can be used for testing date-time fields for a given date. The syntax is:

```
HMIDNT (dtfield, {8|10}, output)
```

where:

dtfield

Is date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

8|10

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for time values down to microseconds.

output

Is the returned date-time value.

Example **Setting the Time to Midnight**

The following request sets the time portion of the DT1 field to midnight in both the 24- and 12-hour systems:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
DT_MID_24/HYYMDS = HMIDNT(STK(1).DT1, 8, DT_MID_24);
DT_MID_12/HYYMDSA= HMIDNT(STK(1).DT1, 8, DT_MID_12);
TYPE "STK(1).DT1 = "STK(1).DT1;
TYPE "DT_MID_24 = <DT_MID_24";
TYPE "DT_MID_12 = <DT_MID_12";
END
```

Syntax **How to Convert the Date Portion of a Date-Time Field to a Date Format**

You can use the new HDATE subroutine in an expression to extract the date portion of a date-time field and convert it to a date format (number of days since the base date 1900/12/31). The syntax is:

```
HDATE (dtfield, output)
```

where:

dtfield

Is the date-time value. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

output

Is the returned date field.

Example **Converting the Date Portion of the DT1 Field to a Date Format**

The following request converts the date portion of the DT1 field to date format YYMD:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
COMPUTE
  DT1_DATE/YYMD = HDATE(STK.DT1, DT1_DATE);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE = <DT1_DATE";
END
```

Syntax **How to Convert a Date Field to a Date-Time Field**

You can use the new HDTTM subroutine in an expression to convert a date field to a date-time field. The time portion is set to midnight. The syntax is:

```
HDTTM (datefield, {8|10}, output)
```

where:

datefield

Is the date value to be converted. You can supply the name of a date field, a date constant, or an expression that returns a date value.

8|10

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for time values down to microseconds.

output

Is the returned date-time value.

Example **Converting a Date Field to a Date-Time Field**

The following request converts the date field DT1_DATE to a date-time field:

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
  COMPUTE
    DT1_DATE/YYMD = HDATE(DT1, DT1_DATE);
    DT2/HYYMDIA  = HDTTM(DT1_DATE, 8, DT2);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "DT1_DATE   = <DT1_DATE";
TYPE "DT2        = <DT2";
END
```

Syntax **How to Convert the Time Portion of a Date-Time Field to a Number**

You can use the new HTIME subroutine in an expression to convert the time portion of a date-time field to a numeric number of milliseconds (if the first argument is 8) or microseconds (if the first argument is 10). For microseconds, the input date-time field must be a 10-byte field. The syntax is:

```
HTIME ({8|10}, dtfield, output)
```

where:

8|10

Is the length of the input date-time value. Use 8 for time values down to milliseconds, 10 for input time values down to microseconds.

dtfield

Is the date-time value to use for extracting the time. You can supply the name of a date-time field, a date-time constant, or an expression that returns a date-time value.

output

Is the returned number of milliseconds or microseconds.

Example **Converting the Time Portion of a Date-Time Field to a Number**

```
MAINTAIN FILE DATETIME
FOR 1 NEXT ID INTO STK;
  COMPUTE MILLISEC/D12.2 = HTIME(8, STK.DT1, MILLISEC);
TYPE "STK(1).DT1 = <STK(1).DT1";
TYPE "MILLISEC   = <MILLISEC";
END
```

Syntax **How to Store the Current Date and Time in a Date-Time Field**

You can use the new HGETC subroutine in an expression store the current date and time in a date-time field. If millisecond or microsecond values are not available in your operating environment, the value returned for these components is zero. The syntax is:

```
HGETC ({8|10}, output)
```

where:

```
8|10
```

Is the length of the returned date-time value. Use 8 for time values down to milliseconds, 10 for input time values down to microseconds.

```
output
```

Is the returned date-time value.

Example Storing the Current Date and Time in a Date-Time Field

```
MAINTAIN
  COMPUTE DT2/HYYMDm = HGETC(10, DT2);
TYPE "DT2          = <DT2";
END
```

Writing Character Expressions

A character expression returns an alphanumeric or text value.

A character expression can consist of the following components, shown below in **bold**:

- An alphanumeric or text constant—that is, a character string enclosed in single or double quotation marks. For example:

```
COMPUTE STATE = 'NY' ;
```

- An alphanumeric or text variable. For example:

```
COMPUTE AddressPartTwo = STATE ;
```

- A function returning an alphanumeric or text result. For example:

```
COMPUTE INITIAL/A1= MASK(FIRSTNAME, '9$$$$$$$$$');
```

- Two or more alphanumeric and/or text expressions combined into a single expression using the concatenation operator. For example:

```
COMPUTE TITLE/A19= 'DR. ' || 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:

Symbol	Represents	Function
	Weak concatenation.	Preserves trailing spaces.
	Strong concatenation.	Suppresses trailing spaces.

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 string's delimiter (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 by the letter n (\n).

When the backslash is used as an escape character, it is not included in the length of the string.

Example 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 = 'This is a "Message"';
```

Example 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/A30 = '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'
```

Example 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";
```

Example 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 `Edwards, C`.

Note that while `Last_Name` has the format `A15`, string 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.

A variable-length character variable's default value 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 string's value 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 *Describing Data With WebFOCUS Language*.

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

```
DECLARE Name/A15 ;
Name = 'Fred Harvey' ;
SAY Name 'End of string\n' ;
```

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)

```
DECLARE Name/A0 ;
Name = 'Fred Harvey' ;
SAY Name 'End of string\n' ;
```

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

```
DECLARE Name/A0 ;
Name = 'Fred Harvey  ' ;
SAY Name 'End of string\n' ;
```

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 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, OMTS, 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 3-44. 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

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.

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'
```

and should instead 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:

Equality	EQ
Inequality	NE
Less than	LT
Greater than	GT
Less than or equal to	LE
Greater than or equal to	GE
Contains the specified character string	CONTAINS
Omits the specified character string	OMITS
Negation	NOT
Conjunction	AND
Disjunction	OR

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

Order	Operators
1	EQ NE LE LT GE GT NOT CONTAINS OMITS
2	AND
3	OR

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 Conditional Expressions

The syntax of a conditional expression is

```
IF boolean THEN {expression1} [ELSE {expression2} ]
```

where:

boolean

Is a Boolean expression. Boolean expressions are described in *Boolean Expressions* on page 3-42.

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 variable's format has been defined:

Data Type	Default value without the MISSING attribute	Default value with the MISSING attribute
Numeric	zero	null
Date and time	space	null
Character	space	null

A null value (sometimes known as missing data) displays as a period (.) by default. You can change the character representation of the null value by issuing the SET NODATA command.

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 *Describing and Accessing Data* 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:

```
COMPUTE target_variable = MISSING;
```

Example Assigning Null Values

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 if 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
```

CHAPTER 4

Form and Control Properties Reference

You determine the appearance and behavior of the forms and controls in your application by setting their properties. Following is an alphabetical list of all of the form and control properties.

You can set some of these properties dynamically at run time using the WINFORM SET command. You can also obtain the value of some of these properties using the WINFORM GET command.

- (GroupCode) Property
- (Name) Property
- Alignment Property
- AlternateRowColor Property
- BackColor Property
- BackColorOver Property
- BackgroundImage Property
- Border Property
- BorderColor Property
- BorderText Property
- BorderWidth Property
- Bottom Property
- CaseStyle Property
- Checked Property
- Columns Property
- Content Property
- CursorPointer Property
- DefaultButton Property
- Enabled Property
- FixedColumns Property
- Font Property
- ForeColor Property
- ForeColorOver Property
- GridLines Property
- HeaderBackColor Property
- HeaderFont Property
- HeaderForeColor Property
- Headers Property
- Height Property
- Help Property
- Hyperlink Property
- Image Property
- ImageDown Property
- ImageOver Property
- ItemBorder Property
- Layer Property
- Left Property
- ListItems Property
- Map Property
- MaximizeBox and MinimizeBox Properties
- MultiSelection Property
- Orientation Property
- Overflow Property
- Password Property
- PenWidth Property
- ReadOnly Property
- Right Property
- Rows Property
- ScrollBars Property
- ScrollHeight and ScrollWidth Properties
- Scrolling Property
- SelectedItem/SelectedItems Property
- Sizeable Property
- Source Property
- Stretched Property
- Tabstop Property
- Text Property
- TextOnLeft Property
- Title Property
- ToolTipText Property
- Top Property
- Validation Property
- Visible Property
- Width Property

(GroupCode) Property

The (GroupCode) property is the name the Form Editor assigns to a collection of grouped controls. You cannot change the value for this property. For more information on grouped controls, see *Grouping Controls* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Grouped controls.

(Name) Property

The (Name) property is the name of the control or form as it is known to your procedure (your end users will not need to know what this name is). The (Name) is different from the title or text that appears on the control, which the end user *does* see.

Tip: We recommend that you give your forms and controls useful names instead of the default. (For example, a button with the text "Add" that inserts data into a data source might be called AddButton.)

Applies to: Forms, all controls.

Alignment Property

The Alignment property determines whether the text in controls is centered, left-justified, or right-justified.

Applies to: Edit boxes, group boxes, multi-edit boxes, text.

Setting dynamically:

```
WINFORM SET form.control.ALIGNMENT TO n;
```

Property Settings:

0 - Left

Aligns text to the left in the selected control (default for edit box, group box, and multi-edit box).

1 - Center

Centers text in the selected control (default for text control).

2 - Right

Aligns text to the right in the selected control.

3 - AlignByType

(Edit box only) Left-justifies character input, and right-justifies numeric input.

AlternateRowColor Property

The AlternateRowColor property enables you to make alternate rows in a grid or HTML Table different colors (useful for “ledger-formatted” output). For example, you could set up your table to have alternating white and green rows. For more information, see *Defining Colors for Your Forms and Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Tip: Avoid using a dithered background color since text is often not as readable when displayed on top of it.

Applies to: Grids, HTML Tables.

Setting dynamically:

```
form.control.SETALTERNATEROWCOLOR(amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets the alternate row color to be the same as BackColor (which means you have no alternate row color).

```
amount_of_red, amount_of_green, amount_of_blue
```

Sets the alternate row to be the color defined by the amount of red, green, and blue.

BackColor Property

The BackColor property specifies the background color of the form or control. For more information, see *Defining Colors for Your Forms and Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Tip: Avoid using a dithered background color since text is often not as readable when displayed on top of it.

Applies to: Forms, buttons, check boxes, combo boxes, edit boxes, grids, HTML Objects, HTML Tables, list boxes, menus, multi-edit boxes, radio buttons, text.

Setting dynamically:

```
form.[control.]SETBACKCOLOR(amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets gray as the background color.

amount_of_red, amount_of_green, amount_of_blue

Sets the background to be the color defined by the amount of red, green, and blue.

BackColorOver Property

The BackColorOver property determines the color of the background when the cursor is on top of a menu.

Tip: Avoid using a dithered background color since text is often not as readable when displayed on top of it.

Applies to: Menus.

Setting dynamically:

```
form.menu. ]SETBACKCOLOROVER(amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets gray as the background color.

amount_of_red, amount_of_green, amount_of_blue

Sets the background to be the color defined by the amount of red, green, and blue.

BackgroundImage Property

The BackgroundImage property specifies an image as a background for a form. To set the value for this property, you use the Image Source dialog box. For more information, see *Using Images* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Setting dynamically: Yes, you can set its value to a Maintain variable.

Applies to: Forms.

Border Property

The Border property specifies the type of border that appears on a control.

Applies to: Edit boxes, frames, group boxes, HTML Objects, HTML Tables, images, menus, multi-edit boxes, radio buttons, text.

Setting dynamically:

```
WINFORM SET form.control.BORDER TO n;
```

Property Settings for edit box, multi-edit box, menu:

0 - No

Displays no border.

1 - Yes

Displays a border (default).

Property Settings for HTML Object, HTML Table, image, radio button, text:

0 - None

Displays no border (default for HTML Object, image, radio button, text).

1 - Normal

Displays a flat border.

2 - 3D

Displays a border with a 3-dimensional effect (default for HTML Table).

3 - Sunken

Displays a shadow on the upper left side of the border that appears to be etched into the form.

4 - Raised

Displays a shadow on the lower right side of the border that appears to be etched into the form.

Property Settings for group box:

0 - 3D

Displays a border with a 3-dimensional effect (default).

1 - Sunken

Displays a shadow on the upper left side of the border that appears to be etched into the form.

2 - Normal

Displays a flat border.

BorderColor Property

The BorderColor property enables you to select a color for a control's borders. For more information, see *Defining Colors for Your Forms and Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Applies to: Group boxes, HTML Objects, HTML Tables, images, radio buttons, text.

Setting dynamically:

```
form.control.SETBORDERCOLOR(amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets black as the border color.

```
amount_of_red, amount_of_green, amount_of_blue
```

Sets the border to be the color defined by the amount of red, green, and blue.

BorderText Property

The BorderText property places a label in the border of a radio button. In order for this label to show up, you must have a border. You can use this property to describe a set of radio buttons.

Applies to: Radio buttons.

Setting dynamically:

```
WINFORM SET form.radiobutton.BORDERTEXT TO "text";
```

BorderWidth Property

The BorderWidth property determines the width of a control's border, measured in pixels.

Applies to: Group boxes, HTML Objects, HTML Tables, images, radio buttons, text.

Setting dynamically:

```
WINFORM SET form.control.BORDERWIDTH TO n;
```

Bottom Property

The Bottom property determines where the bottom of a control is on the form, measured in pixels. Together with the Top, Left, and Right properties, it determines the size of your control and where it is on the form.

Applies to: All controls.

Setting dynamically:

```
WINFORM SET form.control.BOTTOM TO n;
```

CaseStyle Property

The CaseStyle property enables you to convert input in an edit box or multi-edit box to uppercase or lowercase.

Applies to: Edit boxes, multi-edit boxes.

Setting dynamically:

```
WINFORM SET form.control.CASESTYLE TO n;
```

Property Settings:

0 - Unchanged

Displays input as it was originally entered, whether uppercase, lowercase, or mixed (default).

1 - Lower

Converts all input to lowercase.

2 - Upper

Converts all input to uppercase.

Checked Property

The Checked property determines if a check box is initially selected when an end user opens a form. It also indicates whether the end user selected or cleared the check box during run time. You set this value using the *Set Check Box State* dialog box. For more information, see the *Set Check box State Dialog Box* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Check boxes.

Setting dynamically: Yes, you can set its value to a Maintain variable.

Columns Property

For the HTML Table and Grid controls, the Columns property determines which columns of the data source stack you are viewing will appear and in what order. You set the columns using the Control Columns dialog box. For more information, see *Control Columns Dialog Box* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

For the radio button control, the Columns property determines how many columns of radio buttons to draw. The value -1 indicates that WebFOCUS Maintain will place as many columns as can fit in the box. You can also enter any number greater than 0. Together with the Rows property, this property determines the layout of your radio buttons. For more information, see *Determining the Layout of Your Radio Buttons* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Grids, HTML Tables, radio buttons.

Setting dynamically for Grids and HTML Tables: No.

Setting dynamically for radio buttons:

```
WINFORM SET form.control.COLUMNS TO n;
```

Content Property

The Content property determines the HTML code to be inserted inline on the form. You set its value using the HTML Content Source dialog box. For more information, see *HTML Content Source Dialog Box* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: HTML Objects.

Setting dynamically: Yes, you can set its value to a Maintain variable.

CursorPointer Property

The CursorPointer property specifies the type of cursor displayed when an end user moves the cursor over a form or control. For example, it is customary for your cursor to turn into a hand when you move it over a link or a button.

Applies to: Forms, all controls except for combo boxes, frames, and list boxes.

Setting dynamically:

```
WINFORM SET form.[control.]CURSORPOINTER TO n;
```

Property Settings:*0 - Default*

Displays an arrow () when end users move their cursors over the control (default).

1 - IBeam

Displays a text-editing I-beam () when end users move their cursors over the control. This cursor is useful to indicate that end users can enter text into a control, such as an edit box or multi-edit box.

2 - Move

Displays a move cursor ()

3 - Cross

Displays a cursor shaped like a cross ()

4 - Wait

Displays an hourglass cursor () This cursor is useful to indicate that end users need to wait until your application finishes a task.

5 - Help

Displays a cursor with a question mark () This cursor is useful to indicate that end users can get help.

6 - Hand

Displays a hand cursor () This cursor is useful to indicate that end users can accomplish something by clicking the control, such as a Web link or button.

DefaultButton Property

The DefaultButton property enables you to make the selected button control the default when the end user presses the Enter key.

Note: Make sure that you do not have another button first in the tab order. The tab order will override the DefaultButton property.

Applies to: Buttons.

Setting dynamically:

```
WINFORM SET form.button.DEFAULTBUTTON TO {0|1};
```

Property Settings:

0 - No

Disables the default button feature (the default).

1 - Yes

Enables the default button feature.

Enabled Property

The Enabled property specifies whether a control can respond to user generated events. Disabled controls are not part of the tab order.

Applies to: All controls, except for frames.

Setting dynamically:

```
WINFORM SET form.control.ENABLED TO {0|1};
```

Property Settings:

0 - No

Turns off the enable feature and disables the control.

1 - Yes

Turns on the enable feature (the default).

FixedColumns Property

The FixedColumns property determines which columns remain stationary while scrolling through a grid.

Applies to: Grids.

Setting dynamically: No.

Font Property

The Font property enables you to determine the typeface, style, and size of text in controls on the form. You set the font using the Fonts dialog box.

The default form font is MS Sans Serif, Regular, 8 point.

If you set the font for the form, it determines what the default font is for all controls in the form.

You can also set the font for controls individually. If you do not explicitly set the font for a control, it uses the default font, which is determined by the setting for the form.

Applies to: Forms, buttons, check boxes, combo boxes, edit boxes, grids, group boxes, HTML Objects, HTML Tables, list boxes, multi-edit boxes, radio buttons, text.

Setting dynamically: No.

ForeColor Property

The ForeColor property enables you to select the color for text of a control from the Color Palette. For more information, see *Defining Colors for Your Forms and Controls in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

If you change the ForeColor property for the form itself, you set the value for ForeColor for all of the controls on the form that use the Default setting.

Applies to: Forms, buttons, check boxes, combo boxes, edit boxes, grids, group boxes, HTML Objects, HTML Tables, lines, list boxes, menus, multi-edit boxes, radio buttons, text.

Setting dynamically:

```
form.[control.]SETFORECOLOR(amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets the foreground color to be black, or, if it has been set, the form's value for ForeColor.

```
amount_of_red, amount_of_green, amount_of_blue
```

Sets the control to be the color defined by the amount of red, green, and blue.

ForeColorOver Property

The BackColorOver property determines the color of the text when the cursor is on top of a menu.

Applies to: Menus.

Setting dynamically:

```
form.menu. ]SETFORECOLOROVER (amount_of_red, amount_of_green,  
amount_of_blue);
```

Property Settings:

Default

Sets the foreground color to be black, or, if it has been set, the form's value for ForeColor.

amount_of_red, *amount_of_green*, *amount_of_blue*

Sets the background to be the color defined by the amount of red, green, and blue.

GridLines Property

The GridLines property determines whether a grid or HTML Table displays grid lines.

Applies to: Grids, HTML Tables.

Setting dynamically:

```
WINFORM SET form.control.GRIDLINES TO {0|1};
```

Property settings:

0 - No

Hides the grid lines.

1 - Yes

Displays the grid lines (the default).

HeaderBackColor Property

The HeaderBackColor property determines the back color for the header row of a grid or HTML Table. For more information, see *Defining Colors for Your Forms and Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Tip: Avoid using a dithered background color since text is often not as readable when displayed on top of it.

Applies to: Grids, HTML Tables.

Setting dynamically:

```
form.control.SETHEADERBACKCOLOR(amount_of_red, amount_of_green, amount_of_blue);
```

Property Settings:*Default*

Sets gray as the header back color.

```
amount_of_red, amount_of_green, amount_of_blue
```

Sets the header to be the color defined by the amount of red, green, and blue.

HeaderFont Property

The HeaderFont property determines the font for the header row of a grid or HTML Table.

Applies to: Grids, HTML Tables.

Setting dynamically: No.

HeaderForeColor Property

The HeaderForeColor property determines the color of the font in the header row of a grid or HTML Table. For more information, see *Defining Colors for Your Forms and Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Applies to: Grids, HTML Tables.

Setting dynamically:

```
form.control.SETHEADERFORECOLOR(amount_of_red, amount_of_green, amount_of_blue);
```

Property Settings:*Default*

Sets the font color in the header to be black, or, if it has been set, the form's value for ForeColor.

```
amount_of_red, amount_of_green, amount_of_blue
```

Sets the text in the header to be the color defined by the amount of red, green, and blue.

Headers Property

The Headers property determines whether your grid or HTML Table has a header row. You can also remove or add the header row using the Control Columns dialog box. (For more information, see *Control Columns Dialog Box* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.)

Applies to: Grids, HTML Tables.

Setting dynamically:

```
WINFORM SET form.control.HEADERS TO {0|1};
```

Property Settings:

0 - No

Does not display a header row.

1 - Yes

Displays a header row (the default).

Height Property

The Height property determines the height of the form for Windows deployment or, if ScrollBars is turned on, the height of the window the form is displayed in. The height is measured in pixels.

You can also change the Height property by adjusting the size of the form window in the Form Editor.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.

Setting dynamically:

```
WINFORM SET form.HEIGHT TO n;
```

Help Property

The Help property determines the name of Web link associated with this control or form, and an optional location. When the end user presses F1 when the focus is on this form or control, WebFOCUS Maintain opens the link (either at the top or at the optional location).

To set the value for this property, you use the Help dialog box. For more information, see *Assigning Help to Your Forms and Controls* in *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Forms, all controls, except for frames.

Setting dynamically: No.

Hyperlink Property

The Hyperlink property determines whether the end user's browser should treat a text control as a hyperlink. If the browser recognizes the text control as a hyperlink, then it displays a hand whenever end users move their cursors over the text control and it uses its own settings to determine what the text control will look like.

The text control must have an event handler assigned to the Click event; otherwise WebFOCUS Maintain ignores the setting for the Hyperlink property.

For example, in Internet Explorer 4, if you click *Internet Options...* in the View menu, and then click *Colors...* in the General tab, you can set the colors for visited and unvisited links and also turn on the Hover feature. Text controls with the Hyperlink property set to 1 - Yes will then use these settings.

Applies to: Text.

Setting dynamically:

```
WINFORM SET form.text.HYPERLINK TO {0|1};
```

Property Settings:

0 - No

Turns off the Hyperlink property (the default).

1 - Yes

Turns on the Hyperlink property.

Image Property

The Image property determines the name of an image resource to appear on your form. To set the value for this property, you use the Image Source dialog box. For more information, see *Placing Images in Your Forms in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Images.

Setting dynamically: Yes, you can set its value to a Maintain variable.

ImageDown Property

The ImageDown property determines the image that appears when an end user clicks on an image. To set the value for this property, you use the Image Source dialog box. For more information, see *Placing Images in Your Forms in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Images.

Setting dynamically: Yes, you can set its value to a Maintain variable.

ImageOver Property

The ImageOver property determines the image that appears when end users move their mouse cursors over an image. To set the value for this property, you use the Image Source dialog box. For more information, see *Placing Images in Your Forms in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Images.

Setting dynamically: Yes, you can set its value to a Maintain variable.

ItemBorder Property

The ItemBorder property determines whether a menu item in a menu has a border.

Applies to: Menus.

Setting dynamically: No.

Layer Property

The Layer property designates which layer of the form a control is on. The default value is the Default layer. For more information, see *Layering Controls* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Note: This is a form development property only; you cannot use it at run time.

Applies to: All controls.

Setting dynamically: No.

Left Property

For forms in Windows deployment, the Left property determines where the left side of a form is on the screen, measured in pixels.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid for forms. However Version 4 applications that use this property are still supported.

For controls, the Left property determines where the left side of a control is on the form, measured in pixels. Together with the Bottom, Top, and Right properties, it determines the size of your control and where it is on the form.

Applies to: Forms, all controls.

Setting dynamically:

```
WINFORM SET form.[control].LEFT TO n;
```

ListItems Property

The ListItems property determines the items an end user sees in a group of radio buttons, a combo box, or a list box. You set this value using the List Source dialog box. For more information, see *List Source Dialog Box* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Applies to: Radio buttons, combo boxes, list boxes.

Setting dynamically: No.

Map Property

The Map property enables you to define individual parts of an image to be *clickable*. You set this information using the Image Map dialog box. For more information, see *Using Image Maps* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Applies to: Images.

Setting dynamically: No.

MaximizeBox and MinimizeBox Properties

The MaximizeBox and MinimizeBox properties determine whether an end user can maximize or minimize this form in Windows deployment.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.

Setting dynamically: No.

Property Settings:

0 - No

Does not display a maximize or minimize box.

1 - Yes

Displays a maximize or minimize box on the form (the default).

MultiSelection Property

The MultiSelection property specifies whether end users can select multiple items in a list box.

Applies to: List boxes.

Setting dynamically:

```
WINFORM SET form.listbox.MULTISELECTION TO {0|1};
```

Property Settings:

0 - No

Disables multi-selection feature (the default).

1 - Yes

Enables the multi-selection feature.

Orientation Property

The Orientation property determines whether a menu bar is displayed horizontally or vertically.

Applies to: Menus.

Setting dynamically:

```
WINFORM SET form.menu.ORIENTATION To {0|1}
```

Property Settings:

Horizontal (1)

Displays the menu bar horizontally (the default).

Vertical (0)

Displays the menu bar vertically.

Overflow Property

The Overflow property determines how much of an HTML Object or HTML Table appears to an end user. By default, the entire contents of the HTML Object or HTML Table appear on the form. However, if you want to ensure that the HTML Object or HTML Table will never be larger than a certain size, then change the Overflow property to Clip or Scroll.

For example, you would do this if there was information underneath or directly beside the HTML Object or HTML Table and you wanted to ensure that the HTML Object or HTML Table would not overlap the information.

Note: In an HTML Table, the headers will scroll too.

Applies to: HTML Objects, HTML Tables.

Setting dynamically:

```
WINFORM SET form.HTMLcontrol.OVERFLOW TO n;
```

Property Settings:

0 - Visible

Displays all data in an HTML Object or HTML Table (the default).

1 - Clip

Restricts the data to what fits into the control as you drew it. Any information that doesn't fit is not displayed.

2 - Scroll

Restricts the visible data to what fits into the control as you drew it, but places scroll bars on the control (if necessary) so that the end user can view all of the data.

Password Property

The Password property determines what displays when an end user types information into an edit box. If the Password property is on, then the edit box displays asterisks, thus preventing observers from seeing what is being entered into the box.

Applies to: Edit boxes.

Setting dynamically:

```
WINFORM SET form.editbox.PASSWORD TO {0|1};
```

Property Settings:

0 - No

Displays the contents of the edit box (the default).

1 - Yes

Displays asterisks instead of the contents of the edit box.

PenWidth Property

The PenWidth property determines the width of a line, measured in pixels.

Applies to: Lines.

Setting dynamically:

```
WINFORM SET form.line.PENWIDTH TO n;
```

ReadOnly Property

The ReadOnly property determines whether an end user can change the value being displayed in an edit box or multi-edit box.

Applies to: Edit boxes, multi-edit boxes.

Setting dynamically:

```
WINFORM SET form.control.READONLY TO {0|1};
```

Property Settings:

0 - No

Disables the read-only feature (the default).

1 - Yes

Enables the read-only feature.

Right Property

The Right property determines where the right side of a control is on the form. Together with the Top, Bottom, and Left properties, it determines the size of your control and where it is on the form.

Applies to: All controls.

Setting dynamically:

```
WINFORM SET form.control.RIGHT TO n;
```

Rows Property

The Rows property indicates the number of buttons in a column for a radio button. Together with the Columns property, this property determines the layout of your radio buttons.

The default value for this property is -1, meaning that WebFOCUS Maintain will use as many rows as can fit in the radio button box.

For more information, see *Determining the Layout of Your Radio Buttons in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Radio buttons.

Setting dynamically:

```
WINFORM SET form.radiobutton.ROWS TO n;
```

ScrollBars Property

The ScrollBars property determines whether your form has scroll bars in the Form Editor and in Windows deployment. If your form has scroll bars, the end user can see the entire contents of a form even if they do not fit in the available space.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.

Setting dynamically: No.

Property Settings:

0 - No

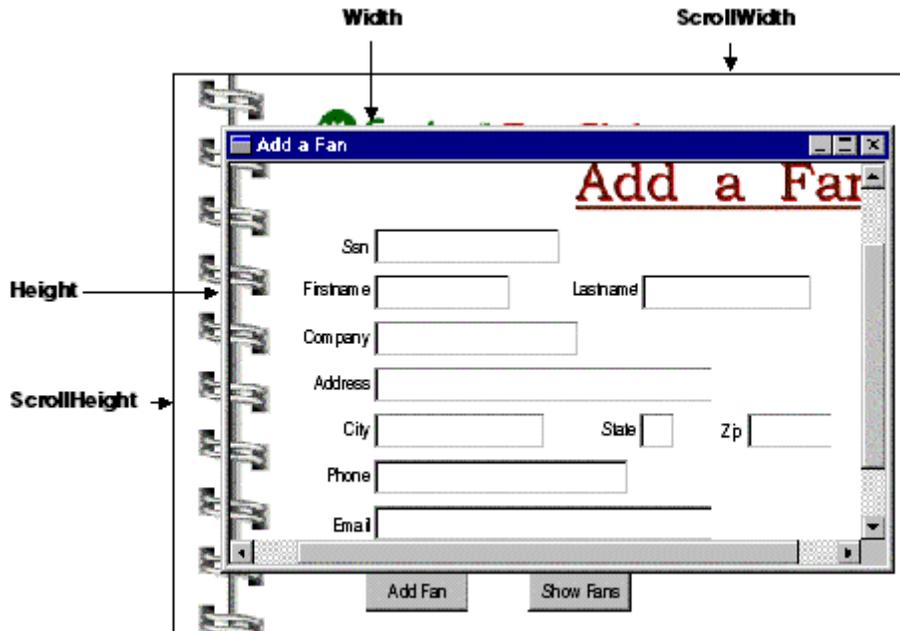
Disables scroll bars.

1 - Yes

Enables scroll bars.

ScrollHeight and ScrollWidth Properties

If you enable ScrollBars, the ScrollHeight and ScrollWidth properties determine the height and width of the entire form, in pixels, for Windows deployment. (The Height and Width properties determines the size of the form that the end user sees.)



If you disable ScrollBars, ScrollHeight and Height are the same, and ScrollWidth and Width are the same.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.

Setting dynamically: No.

Scrolling Property

The Scrolling property determines whether you can scroll in a frame.

Applies to: Frames.

Setting dynamically:

`WINFORM SET form.frame.SCROLLING TO n;`

Property Settings:*0 - Auto*

Enables scroll bars if necessary (the default).

1 - No

Disables scrolling.

2 - Yes

Enables scroll bars.

SelectedItem/SelectedItems Property

The SelectedItem and SelectedItems properties set the value (what the end user selected) from a radio button control, combo box control, or list box control to a Maintain variable or data source stack column. You use the Binding the Selection Result dialog box to set this value. For more information, see *Binding the Selection Result Dialog Box* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

Applies to: Radio buttons, combo boxes, list boxes.**Setting dynamically:** No.

Sizeable Property

The Sizable property determines whether the end user can resize the form in Windows deployment.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.**Setting dynamically:** No.**Property settings:***0 - No*

Does not allow an end user to resize the form.

1 - Yes

Allows an end user to resize the form (default).

Source Property

The Source property determines the contents of a frame. Selecting this property opens the URL Link dialog box. For more information, see *URL Link Dialog Box* in Chapter 6, *Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: Frames.

Setting dynamically: No.

Stretched Property

The Stretched property determines whether you can adjust the size of an image. If you leave this property set to No, the image will be placed on the form in the same size as the original image file.

Applies to: Images.

Setting dynamically:

```
WINFORM SET form.image.STRETCHED TO {0|1};
```

Property settings:

0 - No

Does not allow you to resize the image (the default).

1 - Yes

Enables you to resize the image.

Tabstop Property

The Tabstop property specifies whether an end user can tab to a control. When an end user tabs to a control, it has focus.

You can use the Tab Order dialog box to determine the tab order of controls on a form. For more information, see *Assigning Tab Order to Controls in Developing and Using Controls in Developing WebFOCUS Maintain Applications*.

Applies to: ActiveX controls, buttons, check boxes, combo boxes, edit boxes, list boxes, multi-edit boxes, radio buttons.

Setting dynamically:

```
WINFORM SET form.control.TABSTOP TO {0|1};
```

Property Settings:*0 - No*

Disables the end user from tabbing to the control (the control is removed from tab order).

1 - Yes

Enables the end user to tab to the control (the default).

Text Property

The Text property determines what text appears in a control.

Applies to: Buttons, check boxes, edit boxes, group boxes, multi-edit boxes, text.

Setting dynamically for button, check box, text:

```
WINFORM SET form.control.TEXT TO "text";
```

Setting dynamically for edit box and multi-edit box: Yes, you can set its value to a Maintain variable.

TextOnLeft Property

The TextOnLeft property determines whether the description for a check box appears to the left or the right of the checkbox.

Applies to: Check boxes.

Setting dynamically:

```
WINFORM SET form.checkbox.TEXTONLEFT TO {0|1};
```

Property Settings:*0 - No*

Places the text on the right (default).

1 - Yes

Places the text on the left.

Title Property

The Title property determines the title that appears on your form's title bar at run time. The default is Untitled.

Applies to: Forms.

Setting dynamically:

```
WINFORM SET form.TITLE TO "text";
```

ToolTipText Property

The ToolTipText property enables you to display pop-up information about a form or control when end users place their cursors over the form or control.

Note: Internet Explorer 4 and 5 do not support the ToolTipText property for combo boxes and list boxes. However, since the World Wide Web Consortium spec for DHTML does specify this property, we have left it in the product in case a future release of the browser does support it.

Applies to: Forms, all controls.

Setting dynamically:

```
WINFORM SET form.[control.]TOOLTIPTEXT TO "text";
```

Top Property

For forms in Windows deployment, the Top property determines where the top of a form is on the screen, measured in pixels.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid for forms. However Version 4 applications that use this property are still supported.

For controls, the Top property determines where the top of a control is on the form, measured in pixels. Together with the Bottom, Left, and Right properties, it determines the size of your control and where it is on the form.

Applies to: Forms, all controls.

Setting dynamically:

```
WINFORM SET form.[control.]TOP TO n;
```

Validation Property

The Validation property enables you to formulate rules for checking a value that an end user enters into an edit box. You generate the code for this validation using the Validation Wizard. For more information, see *Using Edit Boxes and Multi-edit Boxes* in *Developing and Using Controls* in *Developing WebFOCUS Maintain Applications*.

This feature is under development.

Applies to: Edit boxes.

Visible Property

The Visible property specifies whether a control is visible or invisible to an end user.

Note: This property can conflict with tab order.

Applies to: All controls.

Setting dynamically:

```
WINFORM SET form.control.VISIBLE TO {0|1};
```

Property Settings:

0 - No

Hides control from the end user.

1 - Yes

Makes controls visible to the end user (default).

Width Property

The Width property determines the width of a form for Windows deployment, measured in pixels, or if ScrollBars is turned on, the width of the window the form is displayed in.

You can also change the Width property by adjusting the size of the form window in the Form Editor.

Note: Because of how WebFOCUS Maintain deploys applications, this property is no longer valid. However Version 4 applications that use this property are still supported.

Applies to: Forms.

Setting dynamically:

```
WINFORM SET form.WIDTH TO n;
```

Width Property

APPENDIX A

WebFOCUS Maintain 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 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 COMPILE'd 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 mentioned 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 and/or master file name.

- (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 care is given to avoiding 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 (via 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.

(FOC03656) Web Cactus Error. Goto Exit Forced. Reason = %1.

(FOC03657) Subscript must be an integer for type %1.

(FOC03658) Invalid type/format for argument #%1 in call 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 Exceeds the Maximum supported name length, currently equal to 31.
- (FOC03685) Unable to Create Execution Manager.
The Execution Manager Object's STARTUP Method failed, 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's stack is not identical to that of CALLEE's.
- (FOC03699) Fatal Error...INTO stack requires HTML column name
For 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's stack contains no columns equivalent to those columns in the CALLEE's stack.
- (FOC03700) FATAL ERROR!... %1 Parameter %2 : Incompatible Types.

(FOC03701) Invalid SET %1 value for MAINTAIN.

Maintain requires the FOCUS fieldname 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.

(FOC03703) Warning. Cannot find the IMPORT file : %1 :.

(FOC03704) Cannot find the file : %1 :.

(FOC03705) 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.

(FOC03706) WINFORM Error on %1: CLOSE not allowed for suspended form.

(FOC03707) WINFORM Error on %1: SHOW command must precede CLOSE.

(FOC03708) WINFORM Error on %1: Duplicate SHOW command for a form.

(FOC03709) WINFORM Error on %1: INACTIVE not allowed for active form.

(FOC03710) WINFORM Error on %1: SHOW command must precede REFRESH.

(FOC03711) WINFORM Error on %1: HIDE is not allowed for active form.

(FOC03712) WINFORM Error on %1: UNHIDE is meaningless for displayed form.

(FOC03713) EDIT CONTROL Error on %1: Associated variable is not an FBIT.

(FOC03714) GRID Error on %1: Associated variable is not a Stack.

(FOC03715) STATIC CONTROL Error on %1: Associated variable is not an FBIT.

(FOC03716) LISTBOX Error on %1: Associated variable is not a Stack.

(FOC03717) RADIOGROUP Error on %1: Associated variable is not a Stack.

(FOC03718) Error on %1: Object is not a Rectangle.

(FOC03719) Error on %1: Object is not a Window.

(FOC03720) Variable %1: is not a stack. It cannot be sorted.

(FOC03721) %1: is not a column in stack %2:. Cannot sort on it.

(FOC03722) Command %1: failed. Out of Memory.

(FOC03723) %1: is not a database field.

The update command operates only on fields.

(FOC03724) 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 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) Comm Fail attempting Connect to Proc %1 at Srvr %2.

(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 a file's cross-references must be on the same sink machine. Check USE list.
- (FOC03757) Acceptable range of values is between %1 and %2.
- (FOC03758) Communications Software Not Installed Or Not In Path.
iWay API 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, please 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 Srvr %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 & 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 which invalidated the parent MAINTAIN's position.
- (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 fieldnames 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.
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 to make sure 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 a method's formal parameter list.
- (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. There 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, please notify the IBI 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 via 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 in question has requirements that conflict with the Application Server architecture. Check subsequent error messages to determine which Maintain is at fault. Most likely cause is FOCUS DB access, DBA in the master or DEFINES in the master.
- (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 Maintains 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 (TABLE, etc.) 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 Maintaine 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 (syntax: MNTCON DEBUG COMPILE procname).
- 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

Index

Symbols

- 3-38
- \$\$ comment delimiter 1-7
- \$* comment delimiter 1-7
- * comment delimiter 1-7
- * multiplication operator 3-3
- * operator 3-3
- *\$ comment delimiter 1-7
- ** exponentiation operator 3-3 to 3-4
- ** operator 3-3
- + operator 3-3 to 3-4
- / division operator 3-3 to 3-4
- \ character in character expressions 3-38 to 3-39
- \n character in character expressions 3-38
- | and || characters in character expressions 3-38

A

- A0 data type 3-40 to 3-41
- A0 variables 2-10
 - passing between procedures 2-10, 2-40, 2-59
- ACTUAL format 3-22
 - date-time data type 3-22
 - relational data sources 3-22 to 3-23
- addition in date expressions 3-13 to 3-14
- addition operator 3-3 to 3-4
- Alignment property 4-2

- ALL
 - COMPUTE command and 2-20
 - COPY command and 2-25
 - DECLARE command and 2-28
 - DELETE command and 2-31
 - INCLUDE command and 2-53
 - NEXT command and 2-66
 - REPEAT command and 2-84
 - REVISE command and 2-92
 - UPDATE command and 2-109
- alphanumeric expressions 3-37
 - backslash () character 3-38 to 3-39
 - concatenating strings 3-38
 - escape character 3-38
 - evaluating 3-38
- alphanumeric format 2-10, 2-59
 - passing A0 variables to procedures 2-10, 2-40, 2-59
 - variable length 3-40 to 3-41
- AlternateRowColor property 4-3
- AND
 - MAINTAIN command and 2-59
 - NEXT command and 2-66
- AND logical operator 3-42 to 3-43
- application window 4-18
- applications 2-45
 - controlling environment 2-100
 - debugging 2-45
- AS keyword in COMPILE command 2-19
- AT
 - CALL command and 2-10
 - EXEC command and 2-40

B

BackColor property 4-3
BackColorOver property 4-4
BackgroundImage property 4-4
backslash () character in character expressions 3-39
BEGIN command 2-8
 nested BEGIN blocks 2-10
Boolean expressions 3-42
Border property 4-5
BorderColor property 4-6
BorderText property 4-6
BorderWidth property 4-6
Bottom property 4-7

C

CALL command 2-10, 2-61
 passing variables between procedures 2-61
CASE command 2-14
 PERFORM command and 2-16
case sensitivity 1-2
 in edit box 4-7
cases 2-14, 2-16
 Top function 2-17
CaseStyle property 4-7
CDN (Continental Decimal Notation) 3-7
 setting in a Maintain procedure 2-104

character expressions 3-37
 backslash () character 3-38 to 3-39
 concatenating strings 3-38
 double quotation marks 3-38
 escape character 3-38
 evaluating 3-38
 single quotation marks 3-38
character format 3-40 to 3-41
 variable length 3-40 to 3-41
character strings 3-38 to 3-39
Checked property 4-7
classes 2-34
 defining 2-34
 libraries 2-64 to 2-65
CLEAR keyword in STACK CLEAR command 2-98
clip setting for Overflow property 4-19
CLOSE keyword in WINFORM command 2-113
CLOSE_ALL keyword in WINFORM command 2-113
color properties 4-3 to 4-4, 4-6, 4-11 to 4-13
Columns property 4-8
commands 1-6
 canceling 1-7
 multi-line 1-6
comments 1-7
 Maintain procedures 1-6 to 1-7
COMMIT command 2-18
COMMIT setting
 setting from a Maintain procedure 2-104
COMPILE command 2-19
COMPUTE command 2-20
 conditional 2-52
 creating user-defined fields 2-23
 IF command and 2-52

- conditional expressions 3-44
- CONTAINS logical operator 3-42 to 3-43
- Content property 4-8
- Continental Decimal Notation (CDN) 3-7
- control properties 4-1 to 4-2
- COPY command 2-24 to 2-25
 - WHERE command and 2-24
- CURRENT keyword in COPY command 2-25
- CursorPointer property 4-8
- Customer Support Service 1-v
 - How to contact 1-v
 - Information required 1-v

D

DATA

- COMPUTE command and 2-20
- DECLARE command and 2-28
- data source paths 2-69
 - reading data from multiple paths 2-69
- data source stacks 2-65
 - libraries and 2-65
 - naming 1-2
- data sources 2-90
 - copying data with NEXT command 2-68
 - DELETE command 2-31
 - INCLUDE command 2-53, 2-56
 - libraries and 2-65
 - NEXT command 2-71
 - REPOSITION command 2-90
 - REVISE command 2-91
 - UPDATE command 2-109, 2-112
- data types 2-16
 - matching in function parameters 2-16
 - synonyms 2-35
- date and time expressions 3-8

- date expressions 3-8
 - addition and subtraction in 3-13 to 3-14
 - components 3-12
 - constants in 3-11
 - evaluating 3-9
 - extracting 3-11
 - formats 3-8, 3-10
 - manipulating in date format 3-11
 - operand format 3-12
- DATEDISPLAY setting 2-104
- date-time data type functions 3-25 to 3-37
 - converting alphanumeric strings 3-27 to 3-28
 - describing 3-14, 3-16
- date-time data types 3-14, 3-17, 3-21 to 3-24
 - ACTUAL attribute 3-22
 - ACTUAL format 3-22 to 3-23
 - assigning 3-24
 - comparing 3-24
 - HOLD data sources 3-22
 - ISO standards 3-27
 - missing values and 3-24
 - SAVE data sources 3-22
 - USAGE 3-17
- date-time formats 3-14
- date-time values 3-14
 - assigning 3-14
- DB2 data sources 3-22 to 3-23
 - date-time data type and 3-22 to 3-23
- DBMS_ERRORCODE 2-102
- DECLARE command 2-28
- DECRYPT command 2-31, 2-36
- DefaultButton property 4-9
- DEFCENT setting 2-104 to 2-105
- DEFINE attribute 2-23
- DELETE command 2-31, 2-33
- DESCRIBE command 2-34

Index

- DFC
 - COMPUTE command and 2-20
 - DECLARE command and 2-28
- DFC setting 2-104
- directory paths 3-38 to 3-39
- DIV operator 3-3 to 3-4
- division operator 3-3 to 3-4
- dollar sign asterisk comment delimiter 1-7
- double dollar sign comment delimiter 1-7
- DROP
 - CALL command and 2-10
 - EXEC command and 2-40
- DROP TABLE command, issuing from Maintain procedure 2-103
- duplicate names 1-2
- E**

- edit boxes 4-7
 - converting input case 4-7
- ELSE
 - IF command and 2-50
- ELSE keyword
 - conditional expressions 3-44
- EMGSRV setting 2-104
- Enabled property 4-10
- ENCRYPT command 2-36
- END command 2-39
- END keyword in GOTO command 2-47
- ENDBEGIN keyword 2-8
- ENDCASE
 - CASE command and 2-14
 - GOTO command and 2-47
- ENDDESCRIBE keyword in DESCRIBE command 2-34
- ENDREPEAT
 - GOTO command and 2-47
 - REPEAT command and 2-84
- ENGINE 2-102 to 2-103
- EQ logical operator 3-42 to 3-43
- error codes, retrieving from DBMSs 2-102
- error messages 2-45
 - displaying 2-45
- escape character 3-38 to 3-39
 - in character expressions 3-38
- EXEC command 2-39
- EXIT keyword in GOTO command 2-47
- EXITREPEAT keyword in GOTO command 2-47
- explicit Top function 2-17
- exponentiation operator 3-3 to 3-4
- expressions 3-1
 - character 3-37
 - conditional 2-52, 3-44
 - date and time 3-8
 - default values 3-44
 - limits in Maintain 3-2 to 3-3
 - logical 3-41
 - null values 3-44
 - numeric 3-3
 - types 3-2
- external procedures 2-39
 - calling 2-39

F

FALSE value for logical expressions 3-42

fields 2-116

- naming 1-2
- null values 2-116, 3-44
- virtual 2-23

FILE keyword in MAINTAIN command 2-59

FILES keyword in MAINTAIN command 2-59

FixedColumns property 4-10

flow of control 2-10

- CALL command 2-10
- CASE command 2-14
- GOTO command 2-46
- IF command 2-50
- looping 2-83
- ON MATCH command 2-78
- ON NEXT command 2-79
- ON NOMATCH command 2-80
- ON NONEXT command 2-81
- PERFORM command 2-82
- REPEAT command 2-83

FOCCOMP ddname/file type 2-19

FocCount variable 2-43

FocCurrent variable 2-10, 2-18, 2-43

FocError variable 2-10, 2-43

FocErrorRow variable 2-10, 2-44

FocFetch variable 2-44

FocIndex variable 2-44

FocMsg stack 2-45

FOCSET command 2-104

FOCUS data sources 2-36

- encrypting 2-36 to 2-37
- restricting existing data sources 2-38

Font property 4-11

FOR keyword 2-109

- COPY command and 2-25
- DELETE command and 2-31
- INCLUDE command and 2-53
- NEXT command and 2-66, 2-69
- REVISE command and 2-92
- UPDATE command and 2-109

ForeColor property 4-11

ForeColorOver property 4-12

format types

- date-time 3-22

forms 2-65

- changing properties dynamically 2-116
- default values 2-116
- displaying at run time 2-113
- libraries and 2-65
- properties 4-1 to 4-2

forms values 2-116

FROM keyword 2-92

- CALL command and 2-10
- COPY command and 2-25
- DELETE command and 2-31
- EXEC command and 2-40
- INCLUDE command and 2-53
- MAINTAIN command and 2-59
- MATCH command and 2-62
- REVISE command and 2-92
- UPDATE command and 2-109

functions

- calling via COMPUTE 2-24
- calling via PERFORM 2-82
- date-time 3-25, 3-27 to 3-37
- date-time component names 3-26 to 3-27, 3-29
- defining 2-14
- libraries 2-64 to 2-65
- naming 1-2
- passing parameters 2-16
- return values 2-17

Index

functions (*continued*)

Top 2-17

G

GE logical operator 3-42 to 3-43

GET keyword in WINFORM command 2-113

global variables 2-30

GOTO command 2-46 to 2-47

data source commands and 2-48

END 2-47

ENDCASE 2-47, 2-49

ENDREPEAT 2-47

EXIT 2-47

EXITREPEAT 2-48

PERFORM command and 2-49, 2-83

TOP 2-47

GridLines property 4-12

GroupCode property 4-2

GT logical operator 3-42 to 3-43

H

H format code 3-16

H USAGE format 3-17

HADD date-time function 3-27, 3-29 to 3-30

HDATE date-time function 3-27, 3-35

HDIFF date-time function 3-27, 3-30 to 3-31

HDTTM date-time function 3-27, 3-35 to 3-36

HeaderBackColor property 4-12

HeaderFont property 4-13

HeaderForeColor property 4-13

Headers property 4-14

Height property 4-14

HELP 4-15

Help property 4-15

HGETC date-time function 3-27, 3-36 to 3-37

HIDE keyword in WINFORM command 2-113

HIGHEST keyword in STACK SORT command 2-99

HMIDNT date-time function 3-34

HNAME date-time function 3-27, 3-31 to 3-32

HOLD data source 3-22

date-time data type 3-22

HPART date-time function 3-27, 3-32

HSETPT date-time function 3-27, 3-33

HTIME date-time function 3-27, 3-36

HYPERLINK 4-15

Hyperlink property 4-15

I

IF command 2-50

BEGIN command and 2-51

conditional COMPUTEs 2-52

conditional expressions 3-44

nesting IF commands 2-51

Image property 4-16

ImageDown property 4-16

ImageOver property 4-16

implicit Top function 2-17

IMPORT keyword in MODULE command 2-64

import modules 2-64

nesting 2-65

INCLUDE command 2-53

data source position 2-56

null values 2-56

unique segments 2-54

INFER command 2-56 to 2-58

INTO keyword 2-66
 CALL command and 2-10
 COPY command and 2-25
 EXEC command and 2-40
 INFER command and 2-57
 MAINTAIN command and 2-59
 MATCH command and 2-62
 NEXT command and 2-66

ItemBorder property 4-16

K

KEEP keyword 2-10
 CALL command and 2-10
 EXEC command and 2-40
 GOTO command and 2-47

L

LANGUAGE setting 2-104
 Language Wizard 2-2
 Layer property 4-17
 LE logical operator 3-42 to 3-43
 Left property 4-17
 libraries 2-64
 importing 2-64
 nesting 2-65
 Top function 2-65
 line feeds in character expressions 3-38
 ListItems property 4-17
 local variables 2-30
 log files 2-97
 SAY command 2-97
 segment and stack values 2-98
 TYPE command 2-105
 logical expressions 3-41
 Boolean expressions 3-42
 evaluating 3-42

operators 3-42 to 3-43
 relational expressions 3-42

logical operators 3-42 to 3-43

loops 2-86
 branching out of 2-90
 ending 2-90
 simple 2-86

LT logical operator 3-42 to 3-43

M

MAINTAIN command 2-59 to 2-60
 calling a procedure from another procedure
 2-61
 specifying data sources 2-60

Maintain commands 2-2

Maintain functions 2-14
 calling via COMPUTE 2-24
 calling via PERFORM 2-82
 defining 2-14
 passing parameters 2-16
 return values 2-17
 Top 2-17

Maintain language 1-1
 assigning values 2-4
 blocks of code 2-2
 canceling commands 1-7
 case sensitivity 1-2
 class libraries 2-7
 command reference 2-1
 comments 1-6 to 1-7
 conditional actions 2-6
 creating objects 2-4
 creating variables 2-4
 defining classes 2-4
 defining functions 2-2
 defining procedures 2-2
 encrypting files 2-3
 forms 2-4
 function libraries 2-7

Index

Maintain language (*continued*)

- loops 2-4
- manipulating stacks 2-4
- messages and logs 2-8
- multi-line commands 1-6
- naming rules 1-2
- procedure contents 1-6
- reading data 2-5
- reserved words 1-4
- rules 1-1
- running procedures 2-3
- transferring control 2-2
- writing transactions 2-6

Maintain language conventions 1-2

Maintain procedures 1-7, 2-96

- blank lines 1-6
- calling 2-39
- comments 1-7
- compiling 2-19, 2-96
- contents 1-6
- encrypting 2-36 to 2-37
- executing 2-96

Map property 4-17

Master Files 2-36

- encrypting 2-36
- performance considerations 2-37

MATCH command 2-62 to 2-64

- NEXT command and 2-70
- REPOSITION command and 2-90

MATCH keyword in ON MATCH command 2-78

MaximizeBox property 4-18

member functions 2-34

- defining with DESCRIBE 2-34

member variables 2-34

- defining with DESCRIBE 2-34

MESSAGE setting 2-104

MinimizeBox property 4-18

minimizing application window at run time 4-18

MISSING attribute 2-20, 2-116, 3-44

- date-time data type and 3-24

MISSING constant 3-44 to 3-45

missing data 2-56

- INCLUDE command 2-56

MISSING keyword 2-20

- COMPUTE command and 2-20
- DECLARE command and 2-28

MOD operator 3-4 to 3-5

modular processing 2-10

MODULE command 2-64

modules 2-64

- importing into procedures 2-64
- nesting 2-65

MSS data source 3-22 to 3-23

- date-time data type and 3-22 to 3-23

multiplication operator 3-3 to 3-4

MultiSelection property 4-18

N

Name property 4-2

names 1-2

- duplicate 1-2
- fully-qualified 1-2
- truncating 1-2
- valid characters 1-2

naming conventions 1-4

- reserved words 1-4

native-mode arithmetic 3-6

NE logical operator 3-42 to 3-43

NEEDS keyword 2-20

- COMPUTE command and 2-20
- DECLARE command and 2-28

nested BEGIN blocks 2-8, 2-10

NEXT command 2-65 to 2-68
 data source navigation 2-71 to 2-73, 2-75
 INTO 2-66
 loading multi-path transaction data 2-69
 MATCH command and 2-70
 multiple rows and 2-69
 path instances and 2-73
 REPOSITION command 2-90
 retrieving rows 2-70
 unique segments 2-77
 WHERE keyword 2-66

NEXT keyword in ON NEXT command 2-79

NODATA parameter
 null representation 3-44
 setting in a Maintain procedure 2-104

NOMATCH keyword in ON NOMATCH command 2-80

NONEXT keyword in ON NONEXT command 2-81

NOT logical operator 3-42 to 3-43

null values 2-116
 COMPUTE command 2-20
 expressions 3-44
 forms 2-116
 INCLUDE command 2-56
 testing 3-45 to 3-46

numeric expressions 3-3
 Continental Decimal Notation (CDN) 3-7
 different operand formats 3-7
 DIV 3-3 to 3-4
 evaluating 3-6
 identical operand formats 3-6
 MOD 3-4
 operators 3-3 to 3-4
 order of evaluation 3-5
 truncating decimal values 3-7

O

OFF keyword 2-28
 COMPUTE command and 2-20
 DECLARE command and 2-28

OMITS logical operator 3-42 to 3-43

ON keyword 2-59
 COMPUTE command and 2-20
 DECLARE command and 2-28
 MAINTAIN command and 2-59
 TYPE command and 2-106

ON MATCH command 2-78

ON NEXT command 2-79

ON NOMATCH command 2-80

ON NONEXT command 2-81

operand format for dates 3-12

OR logical operator 3-42 to 3-43

Oracle data source 3-22 to 3-23
 date-time data type and 3-22 to 3-23

Orientation property 4-19

Overflow property 4-19

P

parameters 2-16

PASS setting 2-104 to 2-105

Password property 4-20

passwords

displaying asterisks instead of contents 4-20
 setting for remote WebFOCUS Servers 2-103

PenWidth property 4-20

PERFORM command 2-16, 2-82
 data source commands and 2-83
 GOTO command and 2-49, 2-83
 nesting 2-83

Index

- PREMATCH setting 2-101
- presentation logic 2-113
 - WINFORM command 2-113
- procedures, Maintain 1-7, 2-96
 - blank lines 1-6
 - calling 2-39
 - comments 1-7
 - compiling 2-19, 2-96
 - encrypting 2-36 to 2-37
 - executing 2-96
- properties
 - changing dynamically 2-116
- properties for forms and controls 4-1 to 4-2
 - Alignment 4-2
 - AlternateRowColor 4-3
 - BackgroundImage 4-4
 - Border 4-5
 - BorderColor 4-6
 - BorderText 4-6
 - BorderWidth 4-6
 - Bottom 4-7
 - CaseStyle 4-7
 - Checked 4-7
 - Columns 4-8
 - Content 4-8
 - CursorPointer 4-8
 - DefaultButton 4-9
 - Enabled 4-10
 - Font 4-11
 - ForeColor 4-11
 - GridLines 4-12
 - GroupCode 4-1 to 4-2
 - HeaderBackColor 4-12
 - HeaderFont 4-13
 - HeaderForeColor 4-13
 - Headers 4-14
 - Height 4-14
 - Help 4-15
 - Hyperlink 4-15
 - Image 4-16
 - properties for forms and controls (*continued*)
 - ImageDown 4-16
 - ImageOver 4-16
 - Layer 4-17
 - Left 4-17
 - ListItems 4-17
 - Map 4-17
 - MaximizeBox 4-18
 - MinimizeBox 4-18
 - MultiSelection 4-18
 - Name 4-2
 - Overflow 4-19
 - Password 4-20
 - PenWidth 4-20
 - ReadOnly 4-20
 - Right 4-21
 - Rows 4-21
 - ScrollBars 4-21
 - ScrollHeight 4-22
 - Scrolling 4-22
 - ScrollWidth 4-22
 - SelectedItems 4-23
 - SetAlternateRowColor property 4-3
 - SetBackColor 4-3
 - SetBorderColor 4-6
 - Sizeable 4-23
 - Source 4-24
 - Stretched 4-24
 - Tabstop 4-24
 - Text 4-25
 - TextOnLeft 4-25
 - Title 4-26
 - ToolTipText 4-26
 - Top 4-26
 - Validation 4-27
 - Visible 4-27
 - Width 4-27

R

ReadOnly property 4-20

records 2-62

- selecting with MATCH command 2-62
- selecting with NEXT command 2-65

REFRESH keyword in WINFORM command 2-113

relational data source 3-22 to 3-23

- date-time data type and 3-22 to 3-23

relational expressions 3-42

REPEAT command 2-83 to 2-84, 2-86, 2-89

- branching out of loops 2-90
- ending loops 2-90
- establishing counters 2-87
- nested loops 2-89
- simple loops 2-86
- UNTIL 2-87
- WHILE 2-87

REPOSITION command 2-90 to 2-91

RESET keyword 2-113

- GOTO command and 2-47
- WINFORM command and 2-113

RESTRICT command 2-38

return values for functions 2-17

RETURNS keyword in CASE command 2-14

REVISE command 2-91 to 2-94

Right property 4-21

ROLLBACK command 2-94 to 2-95

rounding numeric values 3-7

Rows property 4-21

RUN command 2-96

S

SAVE data source 3-22

- date-time data type 3-22

SAY command 2-97 to 2-98

- choosing between SAY and TYPE commands 2-98

scroll setting for Overflow property 4-19

ScrollBars property 4-21

ScrollHeight property 4-22

Scrolling property 4-22

ScrollWidth property 4-22

SEG prefix with SAY command 2-98

segments 2-77

- DELETE command 2-33
- INCLUDE command 2-54
- NEXT command 2-77
- REVISE command 2-91

SelectedItem properties 4-23

SET keyword in WINFORM command 2-113

SET parameters

- NODATA 3-44
- NODATA representation 2-116
- PREMATCH 2-101 to 2-102
- setting in Maintain procedure with
SYS_MGR.FOCSET 2-104

SET_PREMATCH setting 2-101 to 2-102

SetAlternateRowColor property 4-3

SetBackColor property 4-3

SetBackColorOver command 4-4

SetBorderColor property 4-6

SetForeColor property 4-11

SetForeColorOver command 4-12

Index

- SHOW keyword in WINFORM command 2-113
- SHOW_ACTIVE keyword in WINFORM command 2-113
- SHOW_AND_EXIT keyword in WINFORM command 2-113
- SHOW_INACTIVE keyword in WINFORM command 2-113
- Sizeable property 4-23
- SOME keyword 2-20
 - COMPUTE command and 2-20
 - DECLARE command and 2-28
- SORT keyword in STACK SORT command 2-99
- Source property 4-24
- SQL Passthru from Maintain procedures 2-102 to 2-103
- SQL/DS data source 3-22 to 3-23
 - date-time data type and 3-22 to 3-23
- STACK CLEAR command 2-98 to 2-99
- STACK keyword in COPY command 2-25
- STACK prefix with SAY command 2-98
- STACK SORT command 2-99 to 2-100
- stack variables 2-44
 - FocCount 2-43
 - FocIndex 2-44
- stacks 2-23
 - clearing 2-98
 - COPY command 2-24
 - copying rows 2-24
 - creating non-data source columns 2-58
 - INFER command 2-56
 - manipulating 2-4
 - naming 1-2
 - rows 2-99
 - STACK CLEAR command 2-98
 - STACK SORT command 2-99
 - virtual columns 2-23
- Stretched property 4-24
- strong concatenation 3-38
- substrings
 - extracting 3-38 to 3-39
- subtraction in date expressions 3-13 to 3-14
- subtraction operator 3-3 to 3-4
- Sybase data source 3-22 to 3-23
 - and date-time data type 3-22 to 3-23
- synonyms for data types 2-35
- SYS_MGR global object 2-100
 - retrieving error codes 2-102
 - setting prematch 2-101
 - setting WebFOCUS Server environment 2-104
 - SQL Passthru 2-102 to 2-103
- system variables 2-43
 - FocCurrent 2-43
 - FocError 2-43
 - FocErrorRow 2-44
 - FocFetch 2-44

T

- Tabstop property 4-24
- TAKES in CASE command 2-14
- temporary columns 2-23

- temporary fields 2-23
 - text expressions 3-37
 - text format 2-40
 - passing variables to procedures 2-10, 2-40
 - variable length 3-40 to 3-41
 - Text property 4-25
 - TextOnLeft property 4-25
 - THEN keyword 3-44
 - conditional expressions 3-44
 - IF command and 2-50
 - time expressions 3-8
 - Title property 4-26
 - TO in SAY command 2-97
 - ToolTipText property 4-26
 - Top function 2-17
 - libraries and 2-64 to 2-65
 - Top property 4-26
 - TRACEOFF setting 2-104
 - TRACEON setting 2-104
 - TRACEUSER setting 2-104
 - transaction data sources 2-68
 - transaction processing 2-18
 - broadcast rollback 2-95
 - COMMIT command 2-18
 - DELETE command 2-31
 - INCLUDE command 2-53
 - multiple DBMSs rollback 2-95
 - reading multiple-path data sources 2-69
 - REVISE command 2-91
 - ROLLBACK command 2-94
 - UPDATE command 2-109
 - transaction variables 2-43
 - FocCurrent 2-10, 2-43
 - FocError 2-10, 2-43
 - FocErrorRow 2-10, 2-44
 - FocFetch 2-44
 - UPDATE command 2-111
 - transactions 2-113
 - collecting values 2-113
 - troubleshooting 2-45
 - displaying WebFOCUS Server messages 2-45
 - TRUE value for logical expressions 3-42
 - TX data type 3-40 to 3-41
 - TYPE command 2-105 to 2-107
 - choosing between SAY and TYPE commands 2-98
 - embedding spacing information 2-107
 - including variables in a message 2-107
 - justifying variables 2-108
 - multi-line message strings 2-108
 - truncating 2-108
 - writing information to a file 2-108
- ## U
-
- UNHIDE keyword in WINFORM command 2-113
 - unique segments
 - DELETE command 2-33
 - INCLUDE command 2-54
 - NEXT command 2-77
 - UPDATE command 2-112
 - UNTIL keyword in REPEAT command 2-84
 - UPDATE command 2-109 to 2-111
 - data source position 2-112
 - stacks and 2-112
 - transaction variables 2-111
 - unique segments 2-112
 - USAGE attribute 3-17

Index

user ID for remote WebFOCUS Server, setting from Maintain procedure 2-103

USER setting 2-104

V

Validation property 4-27

variable-length character format 2-40
 passing A0 variables to procedures 2-59
 passing variables to procedures 2-10, 2-40

variable-length strings 3-40 to 3-41

variables 2-20
 assigning values with COMPUTE 2-20
 declaring with COMPUTE command 2-20
 declaring with DECLARE command 2-28
 local vs. global 2-30

Visible property 4-27

W

WARNING setting 2-104

weak concatenation 3-38

WebFOCUS Servers
 displaying messages 2-45

WHERE keyword 2-66
 COPY command and 2-25
 NEXT command and 2-66

WHILE keyword in REPEAT command 2-84

Width property 4-27

windows 4-18
 maximizing at run time 4-18
 minimizing at run time 4-18

WINFORM command 2-113, 2-115

 CLOSE 2-113
 CLOSE_ALL 2-113
 GET 2-113
 HIDE 2-113
 REFRESH 2-113
 RESET 2-113
 SET 2-113
 SHOW[_ACTIVE] 2-113
 SHOW_INACTIVE 2-113
 STOP 2-113
 UNHIDE 2-113

wizards 2-2
 Language Wizard 2-2

Y

YRT keyword 2-20
 COMPUTE command and 2-20
 DECLARE command and 2-28

YRTHRESH setting 2-104

Reader Comments

In an ongoing effort to produce effective documentation, the Documentation Services staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please use this form to relay suggestions for improving this publication or to alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

Mail: Documentation Services - Customer Support
Information Builders, Inc.
Two Penn Plaza
New York, NY 10121-2898

Fax: (212) 967-0460

E-mail: books_info@ibi.com

Web form: <http://www.informationbuilders.com/bookstore/derf.html>

Name: _____

Company: _____

Address: _____

Telephone: _____ Date: _____

E-mail: _____

Comments:

Reader Comments