

FOCUS for Mainframe

Maintaining Databases

Version 7.3

EDA, 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 iWay and iWay Software are trademarks of Information Builders, Inc.

Due to the nature of this material, this document refers to numerous hardware and software products by their trademarks. 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 describes FOCUS data management facilities and environments for FOCUS Version 7.3. It is intended for database administrators, application developers, and other information technology professionals who will be creating, restructuring, or directly editing FOCUS and Fusion data sources. This manual is part of the FOCUS documentation set.

References to MVS™ apply to all supported versions of the OS/390®, z/OS™, and MVS operating environments. References to VM apply to all supported versions of the VM/ESA® and z/VM™ operating environments.

- The documentation set consists of the following components:
- The Creating Reports manual describes FOCUS Reporting environments and features.
- The Describing Data manual explains how to create the metadata for the data sources that your FOCUS procedures will access.
- The Developing Applications manual describes FOCUS Application Development tools and environments.
- The Maintaining Databases manual describes FOCUS data management facilities and environments.
- The Using Functions manual describes internal functions and user-written subroutines.
- The Overview and Operating Environments manual contains an introduction to FOCUS and FOCUS tools and describes how to use FOCUS in the VM/CMS and MVS (OS/390) environments.

The users' documentation for FOCUS Version 7.3 is organized to provide you with a useful, comprehensive guide to FOCUS.

Chapters need not be read in the order in which they appear. Though FOCUS facilities and concepts are related, each chapter fully covers its respective topic. To enhance your understanding of a given topic, references to related topics throughout the documentation set are provided. The following pages detail documentation organization and conventions.

How This Manual Is Organized

This manual includes the following chapters:

Chapter/Appendix		Contents
1	Introduction to Maintain	Provides an overview of Maintain, FOCUS's premier facility for developing database transaction applications.
2	Maintain Concepts	Explains essential Maintain concepts including stacks and set-based processing, Winforms and event-driven processing, transaction integrity, and classes and objects.
3	Tutorial: Coding a Procedure	Demonstrates how to code a simple Maintain procedure using basic Maintain logic to read and write to a data source.
4	Tutorial: Painting a Procedure	Demonstrates how to develop an event-driven Maintain procedure, including forms, using the Winform Painter.
5	Using the Winform Painter	Describes how to use the Winform Painter to develop Maintain procedures, including Winforms.
6	Language Rules Reference	Describes the basic rules for using the Maintain language, including rules for naming, adding comments, terminating commands, and continuing commands onto multiple lines.
7	Command Reference	Describes the Maintain language's commands and system variables, including syntax definitions and examples.
8	Expressions Reference	Documents each type of expression in the Maintain language, including operators and rules.
9	Modifying Data Sources With MODIFY	Describes all Maintain built-in functions, including arguments and return values.
10	Designing Screens With FIDEL	Describes how to create full-screen data entry forms for MODIFY and Dialogue Manager procedures.
11	Creating and Rebuilding Databases	Documents how to create and restructure FOCUS and Fusion data sources using the CREATE and REBUILD commands.

Chapter/Appendix		Contents
12	Directly Editing FOCUS Databases With SCAN	Describes how to use SCAN, an interactive line editor, to edit FOCUS data sources.
13	Directly Editing FOCUS Databases With FSCAN	Describes how to use FSCAN, an interactive full-screen editor, to edit FOCUS data sources.
A	Master Files and Diagrams	Contains Master Files and diagrams of sample data sources used in the documentation examples.
B	Error Messages	Describes how to access FOCUS error messages.

Summary of New Features

The FOCUS for Mainframe documentation describes the following new features and enhancements:

New Feature	Manual	Chapter
Increased ACROSS values (from 95)	<i>Creating Reports</i>	Chapter 4, <i>Sorting Tabular Reports</i>
IN-RANGES-OF	<i>Creating Reports</i>	Chapter 4, <i>Sorting Tabular Reports</i>
SET BYDISPLAY	<i>Creating Reports</i>	Chapter 4, <i>Sorting Tabular Reports</i>
Extensions to FORECAST	<i>Creating Reports</i>	Chapter 6, <i>Creating Temporary Fields</i>
Multivariate Regress	<i>Creating Reports</i>	Chapter 6, <i>Creating Temporary Fields</i>
Summary Prefix Operators	<i>Creating Reports</i>	Chapter 7, <i>Including Totals and Subtotals</i>
AnV (VARCHAR) support	<i>Creating Reports</i> <i>Describing Data</i> <i>Using Functions</i>	Chapter 8, <i>Using Expressions</i> Chapter 4, <i>Describing an Individual Field</i> Chapter 4, <i>Character Functions</i>

New Feature	Manual	Chapter
Increased IF-THEN-ELSE	<i>Creating Reports</i>	Chapter 8, <i>Using Expressions</i>
FOCFIRSTPAGE/&FOCNEXTPAGE	<i>Creating Reports</i>	Chapter 9, <i>Customizing Tabular Reports</i>
Increased Number of sort headings/footings	<i>Creating Reports</i>	Chapter 9, <i>Customizing Tabular Reports</i>
Increased column title space	<i>Creating Reports</i>	Chapter 9, <i>Customizing Tabular Reports</i>
Multiple FOLD-LINE	<i>Creating Reports</i>	Chapter 9, <i>Customizing Tabular Reports</i>
NEWPAGE	<i>Creating Reports</i>	Chapter 9, <i>Customizing Tabular Reports</i>
TABLASTPAGE	<i>Creating Reports</i>	Chapter 10, <i>Styling Reports</i>
Stylesheet enhancements	<i>Creating Reports</i>	Chapter 10, <i>Styling Reports</i>
Multiple reports in one PDF file	<i>Creating Reports</i>	Chapter 10, <i>Styling Reports</i>
Decimal Alignment of Headings	<i>Creating Reports</i>	Chapter 10, <i>Styling Reports</i>
Cascading Style Sheets	<i>Creating Reports</i>	Chapter 11, <i>Cascading Style Sheets</i>
Excel 2000	<i>Creating Reports</i>	Chapter 12, <i>Saving and Reusing Report Output</i> Chapter 10, <i>Styling Reports</i>
SET HOLDFORMAT	<i>Creating Reports</i>	Chapter 12, <i>Saving and Reusing Report Output</i>
Excel 97	<i>Creating Reports</i>	Chapter 12, <i>Saving and Reusing Report Output</i> Chapter 10, <i>Styling Reports</i>

New Feature	Manual	Chapter
Holding Missing values	<i>Creating Reports</i>	Chapter 13, <i>Handling Records With Missing Field Values</i>
Compiled Defines	<i>Creating Reports</i>	Chapter 16, <i>Improving Report Processing</i>
FML Hierarchy	<i>Creating Reports</i>	Chapter 17, <i>Creating Financial Reports</i>
FORMULTIPLE	<i>Creating Reports</i>	Chapter 17, <i>Creating Financial Reports</i>
Indenting FML Reports	<i>Creating Reports</i>	Chapter 17, <i>Creating Financial Reports</i>
SET BLANKINDENT	<i>Creating Reports</i>	Chapter 17, <i>Creating Financial Reports</i>
FML Hierarchy	<i>Describing Data</i>	Chapter 4, <i>Describing an Individual Field</i>
Long qualified field names	<i>Describing Data</i>	Chapter 4, <i>Describing an Individual Field</i>
Minus edit format option	<i>Describing Data</i>	Chapter 4, <i>Describing an Individual Field</i>
SUFFIX=TAB	<i>Describing Data</i>	Chapter 5, <i>Describing a Sequential, VSAM, or ISAM Data Source</i>
MDI	<i>Describing Data</i>	Chapter 6, <i>Describing a FOCUS Data Source</i>
GROUPS in FOCUS Files	<i>Describing Data</i>	Chapter 6, <i>Describing a FOCUS Data Source</i>
DATASET for a segment in MFD	<i>Describing Data</i>	Chapter 6, <i>Describing a FOCUS Data Source</i>
XFOCUS Database	<i>Describing Data</i>	Chapter 6, <i>Describing a FOCUS Data Source</i>
Long Segment Names, Long Index Names	<i>Describing Data</i>	Chapter 6, <i>Describing a FOCUS Data Source</i>

New Feature	Manual	Chapter
SET HNODATA	<i>Developing Applications</i>	Chapter 1, <i>Customizing Your Environment</i>
SET HOLDMISS	<i>Developing Applications</i>	Chapter 1, <i>Customizing Your Environment</i>
SET NULL=ON	<i>Developing Applications</i>	Chapter 1, <i>Customizing Your Environment</i>
SET SAVEDMASTERS	<i>Developing Applications</i>	Chapter 5, <i>Enhancing Application Performance</i>
Wide lines	<i>Developing Applications Overview and Operating Environments</i>	Chapter 3, <i>Managing Flow of Control in an Application</i> Chapter 5, <i>CMS Guide to Operations</i> Chapter 6, <i>OS/390 and MVS Guide to Operations</i>
&FOCUSER	<i>Developing Applications</i>	Chapter 3, <i>Managing Flow of Control in an Application</i>
Long Amper variables	<i>Developing Applications</i>	Chapter 3, <i>Managing Flow of Control in an Application</i>
MAINTAIN FILETYPE Extension	<i>Maintaining Databases</i>	Chapter 2, <i>Maintain Concepts</i>
Enhanced screening conditions for Maintain	<i>Maintaining Databases</i>	Chapter 7, <i>Command Reference</i>
COMBINE 63 files	<i>Maintaining Databases</i>	Chapter 9, <i>Modifying Data Sources With MODIFY</i>
FOCUS SETs from Maintain	<i>Maintaining Databases</i>	Chapter 9, <i>Modifying Data Sources With MODIFY</i>
Raised Number of Partitions for External Index	<i>Maintaining Databases</i>	Chapter 11, <i>Creating and Rebuilding Databases</i>
IEDIT	<i>Overview and Operating Environments</i>	Chapter 3, <i>Invoking Your System Editor With IEDIT</i>

New Feature	Manual	Chapter
Relative GDG Number +1	<i>Overview and Operating Environments</i>	Chapter 6, <i>OS/390 and MVS Guide to Operations</i>
SET USERFCHK and SET USERFNS	<i>Using Functions</i>	Chapter 3, <i>Accessing and Calling a Function</i>
FMLINFO	<i>Using Functions</i>	Chapter 10, <i>Numeric Functions</i>
Subroutine NORMSINV and NORMSDST	<i>Using Functions</i>	Chapter 10, <i>Numeric Functions</i>

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), a cross-reference, or an important term. It may also indicate a button, menu item, or dialog box option you can click or select.
this typeface	Highlights a file name or command.
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.

Convention	Description
...	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

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

Customer Support

Do you have questions about FOCUS?

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 FOCUS 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).
- The FOCEXEC procedure (preferably with line numbers).
- Master File with picture (provided by CHECK FILE).
- Run sheet (beginning at login, including call to FOCUS), containing the following information:
 - ? RELEASE
 - ? FDT
 - ? LET
 - ? LOAD
 - ? COMBINE
 - ? JOIN
 - ? DEFINE
 - ? STAT
 - ? SET/? SET GRAPH
 - ? USE
 - ? TSO DDNAME OR CMS FILEDEF
- 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 code, if applicable.
 - Is this related to any other problem?
- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?
- What release of the operating system are you using? Has it, FOCUS, your security system, or an interface system changed?
- Is this problem reproducible? If so, how?

- Have you tried to reproduce your problem in the simplest form possible? For example, if you are having problems joining two data sources, have you tried executing a query containing just the code to access the data source?
- Do you have a trace file?
- How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

User Feedback

In an effort to produce effective documentation, the Documentation Services staff welcomes your opinions 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. Introduction to Maintain	1-1
Using Maintain to Manage Data	1-2
Accessing Data Sources	1-4
Reading From Data Sources	1-4
Writing to Data Sources	1-5
Working With Maintain Procedures	1-5
Developing Procedures	1-5
Storing Procedures	1-6
Executing Procedures	1-7
Maintain Performance	1-7
2. Maintain Concepts	2-1
Set-based Processing	2-2
Which Processes Are Set-based?	2-3
How Does Maintain Process Data in Sets?	2-4
Creating and Defining Database Stacks: An Overview	2-5
Creating a Database Stack	2-6
Defining a Database Stack's Data Source Columns	2-7
Creating a Database Stack's User-defined Columns	2-10
Copying Data Into and Out of a Database Stack	2-11
Referring to Specific Stack Rows Using an Index	2-12
Looping Through a Stack	2-14
Sorting a Stack	2-14
Editing Stack Values	2-15
The Default Database Stack: The Current Area	2-16
Maximizing Database Stack Performance	2-17
Controlling a Procedure's Flow	2-18
Executing Other Maintain Procedures	2-19
Passing Variables Between Procedures	2-20
Accessing Data Sources in the Child Procedure	2-22
Data Source Position	2-22
Optimizing Performance: Data Continuity and Memory Management	2-23
Winforms and Event-driven Processing	2-25
How to Use Winforms	2-26
Designing a Winform	2-27
Using Winforms in Block Mode	2-28
Adjusting Winforms for Your Terminal Configuration	2-28
Optimizing Winform Performance	2-29
Designing Event-driven Applications	2-29
Creating Event-driven Applications	2-29

Reading From a Data Source	2-30
Repositioning Your Location in a Data Source	2-31
Writing to a Data Source	2-32
Evaluating the Success of a Simple Data Source Command	2-32
Evaluating the Success of a Stack-based Write Command	2-33
Transaction Processing	2-34
Why Is Transaction Integrity Important?	2-35
Defining a Transaction	2-36
Canceling a Transaction	2-38
Transactions and Data Source Position	2-38
How Large Should a Transaction Be?	2-38
Designing Transactions That Span Procedures	2-39
Designing Transactions That Span Data Source Types	2-40
When an Application Ends With an Open Transaction	2-41
Determining Whether a Transaction Was Successful	2-41
Concurrent Transaction Processing	2-42
Ensuring Transaction Integrity for FOCUS Data Sources	2-44
Setting COMMIT	2-44
Sharing Access to FOCUS Data Sources	2-45
How the FOCUS Database Server and Change Verification Work	2-46
Selecting Which Segments Will Be Verified for Changes	2-47
Identifying the FOCUS Database Server	2-48
Report Procedures and the FOCUS Database Server	2-49
Sharing Data Sources With Legacy MODIFY Applications	2-49
Ensuring Transaction Integrity for DB2 Data Sources	2-50
Using Transaction Locking to Manage DB2 Row Locks	2-51
Using Change Verification to Manage DB2 Row Locks	2-55
Classes and Objects	2-57
What Are Classes and Objects?	2-57
Class Properties: Member Variables and Member Functions	2-58
Inheritance: Superclasses and Subclasses	2-60
Defining Classes	2-60
Reusing Classes in Class Libraries	2-62
Declaring Objects	2-63
3. Tutorial: Coding a Procedure	3-1
Two Ways to Follow the Tutorial	3-3
Building the Sample Application	3-4
Step 1: Beginning and Ending the Procedure	3-5
Goal	3-5
Methods: MAINTAIN and END Commands	3-5
Solution	3-5

Step 2: Selecting Records	3-6
Goal	3-6
Methods: NEXT and MATCH	3-6
Specifying Data Source Position With the REPOSITION Command	3-7
Solution	3-8
Step 3: Collecting Transaction Values	3-8
Goal	3-8
Methods: WINFORM and NEXT	3-8
Solution	3-9
Step 4: Writing Transactions to the Data Source	3-11
Goal	3-11
Methods: Write Commands, COMMIT, and ROLLBACK	3-11
Solution	3-12
Step 5: Issuing the Procedure	3-13
Goal	3-13
Methods: CALL, COMPILE, RUN	3-13
Solution	3-14
Step 6: Browsing Through a Stack and Using Triggers	3-15
Goal	3-15
Methods: Winform Painter, Triggers, IF, FocIndex, FocCount	3-15
Solution	3-16
Step 7: Displaying and Editing an Entire Stack in a Winform	3-18
Goal	3-18
Methods: Multiple Stacks and Stack Editors (Grids)	3-18
Solution	3-20
4. Tutorial: Painting a Procedure	4-1
Step 1: Creating a New Winform	4-2
Description of the Application	4-2
How to Open the Painter	4-3
Adjusting Winform Appearance	4-3
Naming the Procedure	4-6
Selecting Master Files	4-9
Defining the Winform's Properties	4-11
Winform Title	4-11
Winform Name	4-11
Pop-up Check Box	4-12
Border Check Box	4-12
Stacks	4-12
Source Stacks	4-13
Destination Stacks	4-13
Using the Painter's Menus	4-15

Contents

Saving Your Work and Exiting	4-19
Saving the Procedure	4-19
Exiting the Winform Painter	4-20
Step 2: Adding Fields	4-20
Adding the First Field	4-20
Field	4-22
Object Name	4-22
Source	4-22
Destination	4-23
Length	4-23
Prompt	4-24
Uppercase	4-24
Protected	4-24
OK (PF4)	4-24
Accepts (PF6)	4-24
Color (PF5)	4-25
Cancel (PF3)	4-25
Triggers (PF12)	4-25
Adding Additional Fields	4-27
Editing, Moving, and Resizing Controls	4-27
Step 3: Adding a Grid	4-29
Adding Columns	4-30
Changing Stacks	4-31
Protecting and Unprotecting Columns	4-32
Step 4: Adding Text	4-33
Step 5: Adding Buttons and Triggers	4-35
Adding the First Button and Trigger	4-36
Text	4-36
Justification	4-37
Trigger, Functions	4-37
Shortcut Key Field; PFKeys	4-37
Default Button	4-38
Border	4-38
Object Name	4-39
Adding Additional Buttons and Triggers	4-40
Step 6: Coding Triggers and Other Functions	4-41
Painter-generated Code	4-42
Top Function	4-42
PrevCar Function	4-44
NextCar Function	4-46
GetBody Function	4-46
Step 7: Running the Maintain Request	4-48

5. Using the Winform Painter5-1

- Using the Painter5-2
 - Using Dialog Boxes5-2
 - Using Entry Fields5-3
 - Using List Boxes5-4
 - Using Check Boxes5-5
 - Using the Control Box5-5
 - Using Radio Buttons5-6
 - Using Command Buttons5-7
 - Using Combo Boxes5-8
 - Using Drop Boxes5-9
 - Supporting Colors5-9
 - Supporting Borders5-9
- Files Used by the Winform Painter 5-10
 - How to Access the Painter 5-11
- Saving and Exiting Your Work 5-12
 - Saving the Winform 5-12
- Using the Design Screen 5-14
 - Menu Bar 5-15
 - Procedure Name 5-16
 - Winform Name 5-16
 - Winform Title 5-16
 - Ruler 5-16
 - Canvas 5-16
 - PF Key Help 5-16
- File Menu 5-18
 - New 5-19
 - Open 5-20
 - Save 5-21
 - Save As 5-21
 - Import 5-22
 - Regen 5-24
 - Select Master 5-24
 - Preferences 5-25
 - Exit 5-30
- Edit Menu 5-30
 - Edit Object 5-31
 - Move 5-31
 - Copy 5-32
 - Resize 5-32
 - Delete 5-32

Contents

Forms Menu	5-33
New	5-34
Switch To	5-34
Copy To	5-35
Rename	5-36
Delete	5-36
Properties	5-37
Triggers	5-41
Actions	5-41
Size	5-42
Zoom	5-42
Move	5-43
Objects Menu	5-44
Entry Field	5-45
Text	5-53
Grid	5-54
Browser Functions	5-58
Frame	5-59
Button	5-60
Checkbox	5-64
List Boxes	5-66
Combobox	5-68
Radio Group	5-72
Gen Segment	5-75
Gen Master	5-77
Cases Menu	5-78
Available Cases	5-78
Open	5-79
All	5-80
Copy	5-81
Rename	5-81
Delete	5-82
Done	5-82
Editor	5-82
Help Menu	5-83
Using Triggers, Button Short Cuts, and System Actions	5-83
Specifying Triggers	5-85
Specifying System Actions	5-87

6. Language Rules Reference	6-1
Case Sensitivity	6-2
Specifying Names	6-3
Reserved Words	6-5
What Can You Include in a Procedure?	6-6
Multi-line Commands	6-7
Terminating a Command's Syntax	6-7
Adding Comments	6-8
7. Command Reference	7-1
Language Summary	7-2
Defining a Procedure	7-2
Defining a Maintain Function (a Case)	7-2
Blocks of Code	7-2
Transferring Control	7-3
Executing Procedures	7-4
Encrypting Files	7-5
Loops	7-5
Winforms	7-5
Defining Classes	7-5
Creating Variables	7-5
Assigning Values	7-6
Manipulating Stacks	7-6
Selecting and Reading Records	7-7
Conditional Actions	7-8
Writing Transactions	7-8
Changing the Environment	7-9
Using Libraries of Classes and Functions	7-9
Messages and Logs	7-9
BEGIN	7-10
CALL	7-12
CASE	7-15
Calling a Function: Flow of Control	7-17
Passing Parameters to a Function	7-18
Using a Function's Return Value	7-19
The Top Function	7-19
COMMIT	7-20
COMPILE	7-21
COMPUTE	7-22
Using COMPUTE to Call Functions	7-26
COPY	7-26
DECLARE	7-31
Local and Global Declarations	7-33

Contents

DELETE	7-33
DESCRIBE	7-38
END	7-40
EX	7-41
FocCount	7-42
FocCurrent	7-42
FocError	7-42
FocErrorRow	7-43
FocFetch	7-43
FocIndex	7-43
GOTO	7-44
Using GOTO With Data Source Commands	7-45
GOTO and ENDCASE	7-46
GOTO and PERFORM	7-46
IF	7-47
Coding Conditional COMPUTE Commands	7-50
INCLUDE	7-50
Data Source Position	7-53
Null Data	7-53
INFER	7-54
Defining Non-Data Source Columns	7-55
MAINTAIN	7-56
Specifying Data Sources With the MAINTAIN Command	7-57
Calling a Procedure From Another Procedure	7-58
MATCH	7-59
How the MATCH Command Works	7-61
MNTCON COMPILE	7-61
MNTCON EX	7-62
MNTCON RUN	7-63
MODULE	7-64
What You Can and Cannot Include in a Library	7-64
NEXT	7-65
Subscripted Variables in WHERE Expressions	7-68
Copying Data Between Data Sources	7-70
Loading Multi-Path Transaction Data	7-71
Retrieving Multiple Rows: The FOR Phrase	7-71
Using Selection Logic to Retrieve Rows	7-72
NEXT After a MATCH	7-75
Data Source Navigation Using NEXT: Overview	7-75
Data Source Navigation: NEXT With One Segment	7-76
Data Source Navigation: NEXT With Multiple Segments	7-77
Data Source Navigation: NEXT Following NEXT or MATCH	7-79
Unique Segments	7-82

ON MATCH	7-82
ON NEXT	7-83
ON NOMATCH	7-84
ON NONEXT	7-85
PERFORM	7-87
Using PERFORM to Call Maintain Functions	7-88
Using PERFORM With Data Source Commands	7-88
Nesting PERFORM Commands	7-88
Avoiding GOTO With PERFORM	7-88
RECOMPILE	7-89
REPEAT	7-90
Branching Within a Loop	7-96
REPOSITION	7-97
REVISE	7-98
ROLLBACK	7-101
DBMS Combinations	7-102
RUN	7-102
SAY	7-103
Writing Segment and Stack Values	7-104
Choosing Between the SAY and TYPE Commands	7-104
SET	7-104
STACK CLEAR	7-105
STACK SORT	7-106
SYS_MGR	7-107
SYS_MGR.DBMS_ERRORCODE	7-107
SYS_MGR.ENGINE	7-108
SYS_MGR.FOCSET	7-110
SYS_MGR.PRE_MATCH	7-111
TYPE	7-113
Including Variables in a Message	7-114
Embedding Horizontal Spacing Information	7-114
Embedding Vertical Spacing Information	7-114
Coding Multi-Line Message Strings	7-114
Justifying Variables and Truncating Spaces	7-115
Writing Information to a File	7-115
UPDATE	7-116
Update and Transaction Variables	7-118
Data Source Position	7-119
Unique Segments	7-119
WINFORM	7-120
Managing the Flow of Control in a Winform	7-123
Displaying Default Values in a Winform	7-124
Dynamically Changing Winform Control Properties	7-125

8. Expressions Reference	8-1
Types of Expressions You Can Write	8-2
Expressions and Variable Formats	8-3
Writing Numeric Expressions	8-3
Order of Evaluation	8-5
Evaluating Numeric Expressions	8-6
Identical Operand Formats	8-6
Different Operand Formats	8-7
Continental Decimal Notation	8-7
Writing Date Expressions	8-8
Formats for Date Values	8-8
Evaluating Date Expressions	8-9
Selecting the Format of the Result Variable	8-10
Manipulating Dates in Date Format	8-10
Using a Date Constant in an Expression	8-10
Extracting a Date Component	8-11
Combining Variables With Different Components in an Expression	8-11
Different Operand Date Formats	8-12
Using Addition and Subtraction in a Date Expression	8-13
Writing Date-Time Expressions	8-14
Manipulating Date-Time Values Directly	8-16
Comparing and Assigning Date-Time Values	8-16
Date-Time Subroutines	8-17
Writing Alphanumeric Expressions	8-19
Concatenating Character Strings	8-19
Evaluating Alphanumeric Expressions	8-20
Writing Logical Expressions	8-22
Relational Expressions	8-22
Boolean Expressions	8-23
Evaluating Logical Expressions	8-23
Writing Conditional Expressions	8-25
Handling Null Values in Expressions	8-26
Assigning Null Values: The MISSING Constant	8-26
Conversion in Mixed-Format Null Expressions	8-27
Testing Null Values	8-27

9. Modifying Data Sources With MODIFY9-1

- Introduction9-2
- Examples of MODIFY Processing9-3
 - Adding Data to a Data Source9-3
- Additional MODIFY Facilities9-6
 - Multiple User Access9-8
 - Managing Your Data: Advanced Features 9-11
 - MODIFY Command Syntax 9-13
 - Executing MODIFY Requests 9-14
 - Other Ways of Maintaining FOCUS Data Sources 9-17
 - The EMPLOYEE Data Source 9-18
- Describing Incoming Data 9-19
 - Reading Fixed-Format Data: The FIXFORM Statement 9-20
 - Describing Date Fields 9-29
 - Reading in Comma-delimited Data: The FREEFORM Statement 9-35
 - Identifying Values in a Comma-delimited Data Source 9-37
 - Prompting for Data One Field at a Time: The PROMPT Statement 9-41
- Special Responses 9-46
 - Canceling a Transaction 9-46
 - Ending Execution 9-46
 - Correcting Field Values 9-47
 - Typing Ahead 9-47
 - Repeating a Previous Response 9-48
 - Entering No Data 9-48
 - Breaking Out of Repeating Groups 9-48
 - Invoking the FIDEL Facility: The CRTFORM Statement 9-51
- Entering Text Data Using TED 9-52
 - Entering Text Field Data 9-53
 - Preserving Compatibility of Text Fields Using TED 9-54
 - Defining a Text Field 9-54
 - Displaying Text Fields 9-55
 - Specifying the Source of Data: The DATA Statement 9-56
 - Reading Selected Portions of Transaction Data Sources: The START and STOP Statements 9-57
- Modifying Data: MATCH and NEXT 9-58
 - The MATCH Statement 9-59
 - Adding, Updating, and Deleting Segment Instances 9-64
 - Performing Other Tasks Using MATCH 9-68
 - Modifying Segments in FOCUS Structures 9-71
 - Selecting the Instance After the Current Position: The NEXT Statement 9-88
 - Displaying Unique Segments 9-91

Computations: COMPUTE and VALIDATE	9-92
Computing Values: The COMPUTE Statement	9-93
Validating Transaction Values: The VALIDATE Statement	9-99
VALIDATE Phrases in MATCH and NEXT Statements	9-105
Special Functions	9-107
Reading Cross-Referenced FOCUS Data Sources: The LOOKUP Function	9-110
Messages: TYPE, LOG, and HELPMESSAGE	9-117
Displaying Specific Messages: The TYPE Statement	9-117
Logging Transactions: The LOG Statement	9-126
Displaying Messages: The HELPMESSAGE Attribute	9-131
Displaying Messages: Setting PF Keys to HELP	9-132
Case Logic	9-132
Rules Governing Cases	9-135
Executing a Case at the Beginning of a Request Only: The START Case	9-137
Branching to Different Cases: The GOTO, PERFORM, and IF Statements	9-137
Rules Governing Branching	9-144
GOTO, PERFORM, and IF Phrases in MATCH Statements	9-146
Case Logic Applications	9-148
Tracing Case Logic: The TRACE Facility	9-157
Multiple Record Processing	9-158
The REPEAT Method	9-159
Manual Methods	9-171
Advanced Facilities	9-189
Modifying Multiple Data Sources in One Request: The COMBINE Command	9-190
Differences Between COMBINE and JOIN Commands	9-197
Compiling MODIFY Requests: The COMPILE Command	9-198
Active and Inactive Fields	9-199
Protecting Against System Failures	9-207
Displaying MODIFY Request Logic: The ECHO Facility	9-209
Dialogue Manager Statistical Variables	9-213
MODIFY Query Commands	9-214
Managing MODIFY Transactions: COMMIT and ROLLBACK	9-214
MODIFY Syntax Summary	9-217
MODIFY Request Syntax	9-217
Transaction Statement Syntax	9-219
MATCH and NEXT Statement Actions	9-219
10. Designing Screens With FIDEL	10-1
Introduction	10-2
Using FIDEL With MODIFY	10-2
Using FIDEL With Dialogue Manager	10-4
Screen Management Concepts and Facilities	10-5
Using FIDEL Screens: Operating Conventions	10-6

Describing the CRT Screen	10-7
Specifying Elements of the CRTFORM	10-8
Defining a Field	10-9
Using Spot Markers for Text and Field Positioning	10-12
Specifying Lowercase Entry: UPPER/LOWER	10-14
Data Entry, Display and Turnaround Fields	10-14
Controlling the Use of PF Keys	10-20
Specifying Screen Attributes	10-25
Using Labeled Fields	10-28
Specifying Cursor Position	10-33
Determining Current Cursor Position for Branching Purposes	10-36
Annotated Example: MODIFY	10-38
Annotated Example: Dialogue Manager	10-40
Using FIDEL in MODIFY	10-41
Conditional and Non-Conditional Fields	10-42
Using FIXFORM and FIDEL in a Single MODIFY	10-45
Generating Automatic CRTFORMs	10-47
Using Multiple CRTFORMs: LINE	10-51
CRTFORMs and Case Logic	10-57
Specifying Groups of Fields	10-59
Handling Errors	10-68
Logging Transactions	10-73
Additional Screen Control Options	10-74
Using FIDEL in Dialogue Manager	10-78
Allocating Space on the Screen for Variable Fields	10-78
Starting and Ending CRTFORMs: BEGIN/END	10-79
Clearing the Screen in Dialogue Manager	10-80
Changing the Size of the Message Area: -CRTFORM TYPE	10-81
Annotated Example: -CRTFORM	10-81
Using the FOCUS Screen Painter	10-84
Entering Screen Painter	10-84
Entering Data Onto the Screen	10-88
Identifying Fields: ASSIGN	10-93
Viewing the Screen: FIDEL	10-95
Generating CRTFORMs Automatically	10-96
Terminating Screen Painter	10-97

11. Creating and Rebuilding Databases	11-1
Creating New Databases: The CREATE Command	11-2
Rebuilding Databases: The REBUILD Command	11-4
Controlling the Frequency of REBUILD Messages	11-6
Optimizing File Size: The REBUILD Subcommand	11-8
Changing Database Structure: The REORG Subcommand	11-13
Indexing Fields: The INDEX Subcommand	11-22
Creating an External Index: The EXTERNAL INDEX Subcommand	11-26
Concatenating Index Databases	11-30
Positioning Indexed Fields	11-30
Activating an External Index	11-31
Checking Database Integrity: The CHECK Subcommand	11-33
Confirming Structural Integrity Using ? FILE and TABLEF	11-36
Changing the Database Creation Date and Time: The TIMESTAMP Subcommand	11-39
Converting Legacy Dates: The DATE NEW Subcommand	11-41
How DATE NEW Converts Legacy Dates	11-43
What DATE NEW Does Not Convert	11-44
Using the New Master File Created by DATE NEW	11-45
Action Taken on a Date Field During REBUILD/DATE NEW	11-47
Migrating to a Fusion Database: The MIGRATE Subcommand	11-47
Creating a Multi-Dimensional Index: The MDINDEX Subcommand	11-47
12. Directly Editing FOCUS Databases With SCAN	12-1
Introduction	12-2
SCAN vs. MODIFY, MAINTAIN, HLI, and FSCAN	12-3
Entering SCAN Mode	12-4
Moving Through the Database and Locating Records	12-4
What You See in SCAN Display Lines	12-5
Identifying Data Fields in Scan	12-6
Ways to Move Through Databases	12-6
Displaying Field Names and Field Contents	12-10
Show Lists and Short-Path Records	12-11
Adding Segment Instances	12-12
Moving Segment Instances	12-13
Changing Field Contents	12-13
Deleting Fields and Segments	12-13
Saving Changes Made in SCAN Sessions	12-13
Ending the Session	12-14
Exiting and Saving the Changes	12-14
Exiting Without Saving the Changes	12-14
Auxiliary SCAN Functions	12-14
Displaying a Previous SCAN Subcommand	12-14
Preset X or Y to Execute a SCAN Subcommand	12-15

Subcommand Summary	12-15
AGAIN Command	12-16
BACK Command	12-17
CHANGE Command	12-18
CRTFORM Command	12-20
DELETE Command	12-21
DISPLAY Command	12-22
END Command	12-23
FILE Command	12-23
INPUT Command	12-24
JUMP Command	12-25
LOCATE Command	12-26
MARK Command	12-27
MOVE Command	12-28
NEXT Command	12-29
QUIT Command	12-30
REPLACE Command	12-31
SAVE Command	12-33
SHOW Command	12-33
TLOCATE Command	12-35
TOP Command	12-37
TYPE Command	12-38
UP Command	12-39
X and Y Commands	12-40
? Command	12-41
13. Directly Editing FOCUS Databases With FSCAN	13-1
Introduction	13-2
Databases on Which FSCAN Can Operate	13-2
Segments on Which FSCAN Can Operate	13-3
Fields That FSCAN Can Display	13-3
Database Integrity Considerations	13-4
DBA Considerations	13-4
Entering FSCAN	13-5
Entering FSCAN With a SHOW List	13-5
Allowing Uppercase and Lowercase Alpha Fields	13-7
Using FSCAN	13-7
The FSCAN Facility and FOCUS Structures	13-10
Scrolling the Screen	13-14
Selecting a Specific Instance by Defining a Current Instance	13-16
Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands	13-23
Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands	13-26

Modifying the Database	13-27
Adding New Segment Instances: The "I" Prefix	13-27
Updating Non-Key Field Values	13-29
Changing Key Field Values	13-33
Deleting Segment Instances: The DELETE Command	13-35
Repeating a Command: ? and =	13-37
Saving Changes: The SAVE Without Exiting FSCAN Command	13-37
Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands	13-38
The FSCAN HELP Facility	13-39
Syntax Summary	13-40
Summary of Commands	13-40
Summary of PF Keys	13-48
Summary of Prefix Area Commands	13-48
A. Master Files and Diagrams	A-1
Creating Sample Data Sources	A-2
EMPLOYEE Data Source	A-3
EMPLOYEE Master File	A-5
EMPLOYEE Structure Diagram	A-6
JOBFILE Data Source	A-7
JOBFILE Master File	A-7
JOBFILE Structure Diagram	A-8
EDUCFILE Data Source	A-9
EDUCFILE Master File	A-9
EDUCFILE Structure Diagram	A-10
SALES Data Source	A-11
SALES Master File	A-11
SALES Structure Diagram	A-12
PROD Data Source	A-13
PROD Master File	A-13
PROD Structure Diagram	A-13
CAR Data Source	A-14
CAR Master File	A-15
CAR Structure Diagram	A-16
LEDGER Data Source	A-17
LEDGER Master File	A-17
LEDGER Structure Diagram	A-17
FINANCE Data Source	A-18
FINANCE Master File	A-18
FINANCE Structure Diagram	A-18
REGION Data Source	A-19
REGION Master File	A-19
REGION Structure Diagram	A-19

COURSES Data Source	A-20
COURSES Master File	A-20
COURSES Structure Diagram	A-20
EMPDATA Data Source	A-21
EMPDATA Master File	A-21
EMPDATA Structure Diagram	A-21
EXPERSON Data Source	A-22
EXPERSON Master File	A-22
EXPERSON Structure Diagram	A-22
TRAINING Data Source	A-23
TRAINING Master File	A-23
TRAINING Structure Diagram	A-23
PAYHIST File	A-24
PAYHIST Master File	A-24
PAYHIST Structure Diagram	A-24
COMASTER File	A-24
COMASTER Master File	A-25
COMASTER Structure Diagram	A-26
VIDEOTRK, MOVIES, and ITEMS Data Sources	A-27
VIDEOTRK Master File	A-27
VIDEOTRK Structure Diagram	A-28
MOVIES Master File	A-29
MOVIES Structure Diagram	A-29
ITEMS Master File	A-30
ITEMS Structure Diagram	A-30
VIDEOTR2 Data Source	A-31
VIDEOTR2 Master File	A-31
VIDEOTR2 Structure Diagram	A-32
Gotham Grinds Data Sources	A-33
GGDEMOG Master File	A-33
GGDEMOG Structure Diagram	A-34
GGORDER Master File	A-34
GGORDER Structure Diagram	A-35
GGPRODS Master File	A-35
GGPRODS Structure Diagram	A-36
GGSALES Master File	A-36
GGSALES Structure Diagram	A-37
GGSTORES Master File	A-37
GGSTORES Structure Diagram	A-37

Contents

Century Corp Data Sources	A-38
CENTCOMP Master File	A-39
CENTCOMP Structure Diagram	A-39
CENTFIN Master File	A-40
CENTFIN Structure Diagram	A-40
CENTHR Master File	A-41
CENTHR Structure Diagram	A-43
CENTINV Master File	A-44
CENTINV Structure Diagram	A-44
CENTORD Master File	A-45
CENTORD Structure Diagram	A-46
CENTQA Master File	A-47
CENTQA Structure Diagram	A-48
CENTGL Master File	A-48
CENTGL Structure Diagram	A-49
CENTSYSF Master File	A-49
CENTSYSF Structure Diagram	A-49
CENTSTMT Master File	A-50
CENTSTMT Structure Diagram	A-51
B. Error Messages	B-1
Accessing Error Files	B-2
Displaying Messages Online	B-3

CHAPTER 1

Introduction to Maintain

Topics:

- Using Maintain to Manage Data
- Accessing Data Sources
- Working With Maintain Procedures
- Maintain Performance

Maintain is FOCUS's premier facility for maintaining data sources. It has been designed to meet the needs of the FOCUS community for building sophisticated and robust applications reliably and efficiently.

Application developers who are new to Maintain but familiar with MODIFY should note that Maintain replaces MODIFY as FOCUS's main data source maintenance tool. You will recognize some of the basic commands, and will be pleased to note the extensions and changes that put increased power at your fingertips.

Using Maintain to Manage Data

Maintain combines power and simplicity in a single data management facility. It incorporates the following features:

- **Object-oriented development.** You can define classes and declare objects that model the real-world entities, such as customers and business transactions, with which your enterprise deals. Object-oriented development saves you time by making it easier for you to reuse source code by means of property inheritance and class libraries, and enables you to leverage the powerful Maintain language by creating your own data types.
- **Set-based processing.** You can manipulate a group of data source records at the same time. You can define the group as a sequential range of records, as all records that satisfy selection criteria, or a combination of the two. For example, with one simple command you can select and retrieve all of the records for the first 100 employees who have the job code A25.
- **Record-at-a-time processing.** In addition to set-based processing, you can also identify and work with one record at a time.
- **Sophisticated graphical user interface (GUI).** You can use the Winform Painter to create sophisticated interactive forms, called Winforms, for entering data, displaying information, and selecting options. You can design these Winforms to include user-friendly features such as dialog boxes for requesting special information, check boxes for making choices, list boxes for selecting values, buttons for invoking functions, and entry fields with automatic data validation for entering valid values.
- **Triggers and event-driven processing.** The flow of control in conventional processing is mostly pre-determined—that is, the application developer determines the few paths that the end user will be able to take through the procedure.

To make your application more responsive to the end user—using the graphical user interface—Maintain provides triggers and event-driven processing. A trigger is a specified event that invokes or *triggers* a function. Each time that the event (such as an end user pressing a function key) occurs, the function is invoked. For example, you might create a trigger for retrieving data: it would notice whenever an end user presses the PF7 key while in a Winform at run time, and would react by retrieving the specified data from the data source and displaying it in the Winform for the user to edit.

- **Event-driven development.** Developing a procedure by writing out sequential lines of code may be sufficient for conventional linear processing, but event-driven processing demands event-driven development. Developing an application in this way enables you to build much of the application's logic around the user interface. For example, you can develop part of the user interface (a Winform), then assign a trigger to a particular Winform event, then specify the action (that is, the function) associated with the trigger, and finally code the function—all from within the Winform Painter.
- **Flow-of-control.** Maintain provides many different ways of controlling the flow of a procedure, using enhanced versions of commands found in MODIFY as well as entirely new commands and functions. For example, you can transfer control unconditionally to functions, using PERFORM, and to procedures, using CALL; transfer control conditionally using IF; declare functions (also known as cases) with CASE, and simple blocks of code with BEGIN; and loop using REPEAT.
- **Transaction Integrity.** Maintain enables you to define multiple data source operations as a single transaction, and to ensure that the entire transaction is written to the data source only if all of its component operations were successful. Maintain does this by respecting your DBMS's native transaction integrity strategy.
- **Modular processing.** You can create several Maintain procedures which work together, one procedure calling another.

These are just some of the features you can use to develop powerful, flexible, and robust data source management applications.

Accessing Data Sources

To access a data source using Maintain you must identify the data source to:

- **FOCUS.** The standard FOCUS requirements apply: you need a Master File that describes the data source, supplemented—for some types of data sources—by an Access File. For detailed information about Master and Access Files, see the *Describing Data* manual and the FOCUS Interface documentation for the types of data sources that you are accessing.
- **The Maintain procedure.** You identify data sources to a Maintain procedure by specifying them following the FILE keyword in the procedure's MAINTAIN command. For example, if you wish to read records from a data source named AutoTeller, and update a data source named Accounts, the procedure would begin with the following command:

```
MAINTAIN FILES AutoTeller AND Accounts
```

The MAINTAIN command is described in Chapter 7, *Command Reference*.

Reading From Data Sources

Maintain can read many kinds of data sources, including DB2, Teradata, and FOCUS databases, and VSAM and fixed-format sequential data files. Maintain reads data sources using the NEXT and MATCH commands.

Maintain can read data sources to which the operating system does not permit write access.

Maintain can read individual and joined data sources. Maintain supports joins that are defined in the Master File. For information about defining joins in the Master File, see the FOCUS Interface documentation for the types of data sources that you wish to join, and the *Describing Data* manual for FOCUS data sources.

If a database administrator has applied FOCUS DBA security to a data source, and you wish to read:

- **A cross-referenced FOCUS database** joined in the Master File, the DBA specification must grant you access privileges. Any type of access—read (R), write (W), read/write (RW), or update (U)—is sufficient.
- **All other types of data sources**, such as joined host data sources, relational tables joined by a join defined in the Master File, and simple (unjoined) data sources, the DBA specification must grant you access privileges of write (W), read/write (RW), or update (U). Each of these enables you to read from and write to the data source; with regard to Maintain, these three DBA access privileges are equivalent.

Maintain supports DBA except for the VALUE and NOPRINT attributes. See the *Describing Data* manual for information about DBA.

Writing to Data Sources

Maintain can write to many kinds of data sources, including DB2, Teradata, and FOCUS databases, and VSAM data files. Maintain writes to data sources using the INCLUDE, UPDATE, REVISE, and DELETE commands.

If your database administrator has applied FOCUS DBA security to a data source, the DBA specification must grant you access privileges of write (W), read/write (RW), or update (U). Each of these enables you to write to and read from the data source using any Maintain data source command; with regard to Maintain, these three DBA access privileges are equivalent.

Maintain supports DBA except for the VALUE and NOPRINT attributes. See the *Describing Data* manual for information about DBA.

Working With Maintain Procedures

A Maintain application can include multiple procedures. Each procedure comprises one FOCEXEC file and, optionally, a WINFORMS file containing the procedure's Winforms. This section gives you a brief overview of how to develop, store, and run Maintain procedures.

Developing Procedures

FOCUS provides you with several ways to develop a Maintain procedure:

- **Event-driven development.** Most data source procedures are event-driven: a user performs an event—such as entering data, clicking a button, or selecting a list item—and this event triggers some logic. You can develop such procedures most effectively if you follow the same model: first identify an action that the user needs to perform, then develop the user interface to enable it, and finally code the logic that performs the action.

For example, consider a data entry procedure in which the user needs to enter data and write it to the data source. You can create a Winform with entry fields and a command button, then assign a trigger with a function to the button, and finally code the function to update the data source. You perform all of these steps seamlessly within the Winform Painter. The Painter even generates some of the code for you.

- **Process-driven development.** You can develop the procedure's data source logic separately from its user interface logic: code the Maintain procedure using an editor such as TED, and develop the Winforms using the Winform Painter.

Because each Maintain procedure must reside in its own FOCEXEC, you cannot create ad hoc procedures at the FOCUS command prompt. For more information about using the Winform Painter to develop procedures, see Chapter 4, *Tutorial: Painting a Procedure*, and Chapter 5, *Using the Winform Painter*. For more information about using TED, see the *Overview and Operating Environments* manual.

Storing Procedures

Each Maintain procedure is stored in two files:

- **FOCEXEC.** The procedure's business logic is stored in a FOCEXEC file. The file should contain a single Maintain request: it should not contain multiple requests, and should not contain other elements such as Dialogue Manager commands or SET commands. For more information about FOCEXECs, see the *Developing Applications* manual.
- **WINFORMS.** Most of the procedure's presentation logic is stored as Winforms in a WINFORMS file. You create and edit Winforms using the Winform Painter. You should make changes to WINFORMS files using the Painter only, and not attempt to edit them directly: all changes made outside the Painter are lost the next time the file is edited in the Painter. When you use:
 - **CMS,** the file type is WINFORMS. The file's attributes are RECFM=F and LRECL=80. When FOCUS searches for WINFORMS files it uses the standard CMS search order.
 - **MVS,** the file is stored in a PDS allocated to ddname WINFORMS; if you wish you can concatenate several data sets together. If you neglect to allocate the WINFORMS PDS, FOCUS will dynamically allocate it for you. The following lines show sample commands for allocating WINFORMS under TSO:

```
ALLOC F(WINFORMS) DA(WINFORMS.DATA) SHR
```

and under MSO or MVS FOCUS:

```
DYNAM ALLOC FILE WINFORMS DATASET user.WINFORMS.DATA
```

The PDS's DCB attributes are RECFM=FB, LRECL=80, and BLKSIZE, a multiple of LRECL. When FOCUS searches for a WINFORMS file, it first looks for ddname WINFORMS. If ddname WINFORMS is not allocated, FOCUS attempts to allocate the data set prefix.WINFORMS.data to ddname WINFORMS. If the specified member is not there, FOCUS displays an error message.

Each procedure has one FOCEXEC file and—if it has a user interface—one WINFORMS file. The two files share the same file name (under CMS) or member name (under MVS). For example, the Payroll procedure is stored in the files PAYROLL FOCEXEC and PAYROLL WINFORMS.

Executing Procedures

FOCUS provides you with the flexibility of executing a Maintain procedure from different kinds of environments, and employing different control techniques.

In addition, you can accelerate execution by compiling a procedure using the COMPILE command. You may wish to work with the procedure in its regular form while you are developing it, and then—once it is finished—compile it to make it faster.

You can run a procedure from the following environments:

- **The FOCUS command prompt.** You can run a Maintain procedure at the FOCUS command prompt by issuing the EXEC command (for uncompiled procedures) or the RUN command (for compiled procedures).
- **Maintain procedures.** You can run one Maintain procedure from another by issuing the CALL command in the parent procedure. If the child procedure exists in both compiled and uncompiled form, Maintain executes the compiled version. You can pass data between the two procedures.
- **Other types of procedures.** You can run a Maintain procedure from a non-Maintain procedure by issuing the EXEC command (for uncompiled procedures) or the RUN command (for compiled procedures) in the calling procedure. You can issue the command in the calling procedure's FOCEXEC file, or in an Execution window in the calling procedure's FMU or TRF file. When using EXEC you can pass data to the Maintain procedure.

Alternatively, you can run an uncompiled Maintain procedure by naming it in a Menu window (using the Window Painter's FOCEXEC Name option) in the calling procedure's FMU or TRF file.

The EXEC command, Execution and Menu windows, and the Window Painter are all described in the *Developing Applications* manual. The CALL, COMPILE, and RUN commands are described in Chapter 7, *Command Reference*.

Maintain Performance

If you wish to maximize Maintain efficiency, exploit Maintain features. Refrain from coding a Maintain procedure as if you were using the MODIFY language. For example, you will achieve higher performance if you use set-based processing in place of record-at-a-time processing whenever applicable.

Another way to optimize Maintain performance is to keep all Winforms in a single Maintain procedure, reducing memory consumption. This applies mostly to CMS.

Maintain Performance

CHAPTER 2

Maintain Concepts

Topics:

- Set-based Processing
- Controlling a Procedure's Flow
- Executing Other Maintain Procedures
- Winforms and Event-driven Processing
- Reading From a Data Source
- Writing to a Data Source
- Transaction Processing
- Defining a Transaction
- Ensuring Transaction Integrity for FOCUS Data Sources
- Classes and Objects

To exploit fully the potential and productivity of Maintain, you should become familiar with some basic Maintain concepts, including:

- Processing data in sets by using stacks.
- Controlling the flow of a Maintain application and the procedures within it.
- Developing presentation logic using Winforms and triggers.
- Reading and writing to data sources.
- Ensuring transaction integrity.
- Creating classes and objects.

Set-based Processing

Maintain provides the power of set-based processing, enabling you to read, manipulate, and write groups of records at a time. You manipulate these sets of data using a data structure called a database stack.

A database stack is a simple temporary table. Generally, columns in a database stack correspond to data source fields, and rows correspond to records, or path instances, in that data source. You can also create your own “user-defined” columns.

The intersection of a row and a column is called a cell and corresponds to an individual field value. The database stack itself represents a data source path.

For example, consider the following Maintain command:

```
FOR ALL NEXT Emp_ID Pay_Date Ded_Amt INTO PayStack  
WHERE Employee.Emp_ID EQ SelectedEmpID;
```

This command retrieves Emp_ID and the other root segment fields, as well as the Pay_Date, Gross, Ded_Code, and Ded_Amt fields from the Employee data source and holds them in a database stack named PayStack. Because the command specifies FOR ALL, it retrieves all of the records at the same time; you do not need to repeat the command in a loop. Because it specifies WHERE, it retrieves only the records you need—in this case, the payment records for the currently-selected employee.

You could just as easily limit the retrieval to a sequence of data source records, such as the first six payment records that satisfy your selection condition,

```
FOR 6 NEXT Emp_ID Pay_Date Ded_Amt INTO PayStack  
WHERE Employee.Emp_ID EQ SelectedEmpID;
```

or even restrict the retrieval to employees in the MIS department earning salaries above a certain amount:

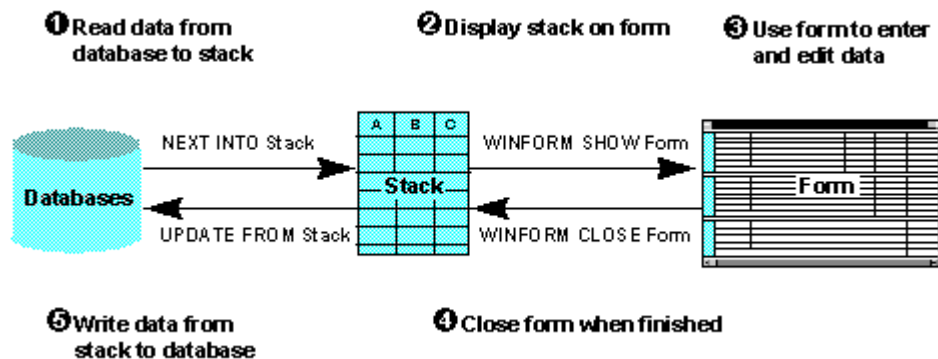
```
FOR ALL NEXT Emp_ID Pay_Date Ded_Amt INTO PayStack  
WHERE (Employee.Department EQ 'MIS') AND  
(Employee.Curr_Sal GT 23000);
```


Which Processes Are Set-based?

You can use set-based processing for the following types of operations:

- **Selecting records.** You can select a group of data source records at one time using the NEXT command with the FOR prefix. Maintain retrieves all of the data source's records that satisfy the conditions you specified in the command and then automatically puts them into the database stack that you specified.
- **Collecting transaction values.** You can use Winforms to display, edit, and enter values for groups of rows. The rows are retrieved from a database stack, displayed in the Winform, and, when the user is finished, placed back into a stack. You can also use the NEXT command to read values from a transaction file into a stack.
- **Writing transactions to the data source.** You can include, update, or delete a group of records at one time using the INCLUDE, UPDATE, REVISE, or DELETE commands with the FOR prefix. The records come from the database stack that you specify in the command.
- **Manipulating stacks.** You can copy a set of records from one database stack to another and sort the records within a stack.

The following diagram illustrates how these operations function together in a procedure:



The diagram is explained in detail below:

1. The procedure selects several records from the data source and, for each record, copies the values for fields A, B, and C into the database stack. It accomplishes this using the NEXT command.
2. The procedure displays a Winform on the screen. The Winform shows multiple instances of fields A, B, and C; the field values shown on the screen are taken from the stack. This is accomplished using the WINFORM SHOW command.
3. The procedure user views the Winform and enters and edits data. As the Winform responds to the user's activity, it automatically communicates with the procedure and updates the stack with the new data.
4. The procedure user clicks a button to exit the Winform; the button accomplishes this by triggering the WINFORM CLOSE command.
5. The procedure writes the values for fields A, B, and C from the stack to the selected records in the data source. The procedure accomplishes this using the UPDATE command.

How Does Maintain Process Data in Sets?

Maintain processes data in sets using two features:

- **The command prefix FOR.** When you specify FOR at the beginning of the NEXT, INCLUDE, UPDATE, REVISE, and DELETE commands, the command works on a group of records, instead of on just one record.
- **Stacks.** You use a database stack to hold the data from a group of data source or transaction records. For example, a stack can hold the set of records that are output from one command (such as NEXT or WINFORM) and provide them as input to another command (such as UPDATE). This enables you to manipulate the data as a group.

Creating and Defining Database Stacks: An Overview

Maintain makes working with stacks easy by enabling you to create and define a database stack dynamically, simply by using it. For example, when you specify a particular stack as the destination stack for a data source retrieval operation, that stack is defined as including all of the fields in all of the segments referred to by the command. Consider the following NEXT command, which retrieves data from the VideoTrk data source into the stack named VideoTapeStack:

```
FOR ALL NEXT CustID INTO VideoTapeStack;
```

Because the command refers to the CustID field in the Cust segment, all of the fields in the Cust segment (from CustID through Zip) are included as columns in the stack. Every record retrieved from the data source is written as a row in the stack.

Example Creating and Populating a Simple Database Stack

If you are working with the VideoTrk data source, and you want to create a database stack containing the ID and name of all customers whose membership expired after June 24, 1992, you could issue the following NEXT command:

```
FOR ALL NEXT CustID INTO CustNames WHERE ExpDate GT 920624;
```

The command does the following:

1. Selects (NEXT) all VideoTrk records (FOR ALL) that satisfy the membership condition (WHERE).
2. Copies all of the fields from the Cust segment (referenced by the CustID field) from the selected data source records into the CustNames stack (INTO).

The resulting CustNames stack looks like this (some intervening columns have been omitted to save space):

CustID	LastName	...	Zip
0925	CRUZ	...	61601
1118	WILSON	...	61601
1423	MONROE	...	61601
2282	MONROE	...	61601
4862	SPIVEY	...	61601
8771	GARCIA	...	61601
8783	GREEN	...	61601
9022	CHANG	...	61601

Creating a Database Stack

You create a database stack:

- **Implicitly**, by specifying it in a NEXT or MATCH command as the destination (INTO) stack, or by associating it in the Winform Painter with a control.

Winforms are introduced in *Winforms and Event-driven Processing* on page 2-25; the Winform Painter used to design and create Winforms is described in Chapter 5, *Using the Winform Painter*.

- **Explicitly**, by specifying it in an INFER command.

For example, this NEXT command creates the EmpAddress stack:

```
FOR ALL NEXT StreetNo INTO EmpAddress;
```

Defining a Database Stack's Data Source Columns

When you define a database stack, you can include any field in a data source path. Maintain defines a stack's data source columns by performing the following steps:

1. Scanning the procedure to identify all of the NEXT, MATCH, and INFER commands that use the stack as a destination and all the controls that use the stack as a source or destination.
2. Identifying the data source fields that these commands and controls move into or out of the stack:
 - **NEXT commands** move the fields in the data source field list and WHERE phrase.
 - **MATCH commands** move the fields in the data source field list.
 - **INFER commands** move all the fields specified by the command.
 - **Controls** move all the fields specified by the control.
3. Identifying the data source path that contains these fields.
4. Defining the stack to include columns corresponding to all the fields in this path.

You can include any number of segments in a stack, as long as they all come from the same path. When determining a path, unique segments are interpreted as part of the parent segment. The path can extend through several data sources that have been joined together. Maintain supports joins that are defined in the Master File. For information about defining joins in the Master File, see the FOCUS Interface documentation for the types of data sources that you wish to join, and the *Describing Data* manual for FOCUS data sources. (Maintain can read data sources that have been joined to a host data source, but cannot write to them.)

The highest specified segment is known as the anchor and the lowest specified segment is known as the target. Maintain creates the stack with all of the segments needed to trace the path from the root segment to the target segment:

- It automatically includes all fields from all of the segments in the path that begin with the anchor and continue to the target.
- If the anchor is not the root segment, it automatically includes the key fields from the anchor's ancestor segments.

Example **Defining Data Source Columns in a Database Stack**

In the following source code, a NEXT command refers to a field (Last_Name) in the EmpInfo segment of the Employee data source, and reads that data into EmpStack; another NEXT command refers to a field (Salary) in the PayInfo segment of Employee and also reads that data into EmpStack:

```
NEXT Last_Name INTO EmpStack;  
. . .  
FOR ALL NEXT Salary INTO EmpStack;
```

Based on these two NEXT commands, Maintain defines a stack named EmpStack, and defines it as having columns corresponding to all of the fields in the EmpInfo and PayInfo segments:

Emp_ID	Last_Name	...	Ed_Hrs	Dat_Inc	...	Salary	JobCode
071382660	STEVENS	...	25.00	82/01/01	...	\$11,000.00	A07
071382660	STEVENS	...	25.00	81/01/01	...	\$10,000.00	A07

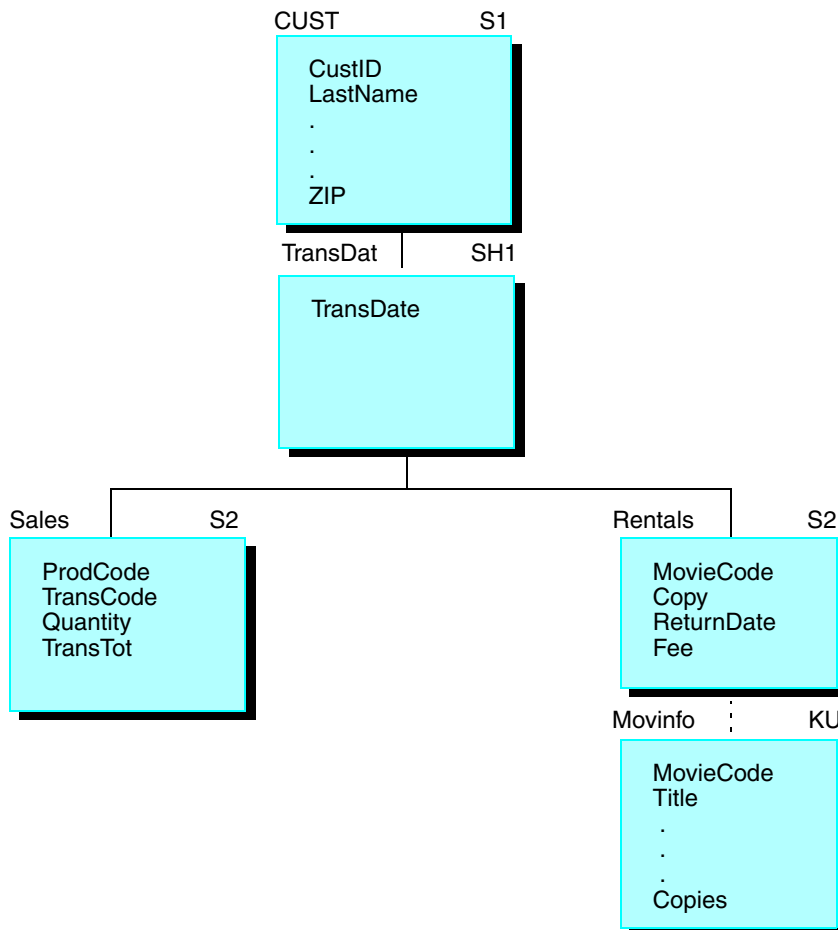
Example Establishing a Path Using Keys and Anchor and Target Segments

The following code populates CustMovies, a database stack that contains video rental information for a given customer. The first NEXT command identifies the customer. The second NEXT command selects a field (TransDate) from the second segment and a field (Title) from the bottom segment of a path that runs through the joined VideoTrk and Movies data sources:

```
NEXT CustID WHERE CustID IS '7173';

FOR ALL NEXT TransDate Title INTO CustMovies
  WHERE Category IS 'COMEDY';
```

The structure of the joined VideoTrk and Movies data sources looks like this:



In this NEXT command, the TransDat segment is the anchor and the MovInfo segment is the target. The resulting CustMovies stack contains all the fields needed to define the data source path from the root segment to the target segment:

- The anchor's ancestor segment, Cust (key field only).
- All segments from the anchor through the root: TransDat, Rentals, MovInfo (all fields).

The stack looks like this:

CustID	TransDate	MovieCode	...	Title	...	Copies
7173	91/06/18	305PAR	...	AIRPLANE	...	2
7173	91/06/30	651PAR	...	MY LIFE AS A DOG	...	3

Creating a Database Stack's User-defined Columns

In addition to creating database stack columns that correspond to data source fields, you can also create database stack columns that you define yourself. You can define these columns in two ways:

- **Within a procedure.** You can create a stack column (as well as a user-defined field) by issuing a COMPUTE command. You can also use the COMPUTE command to assign values to stack cells.

Because all Maintain variables are local to a procedure, you must redefine variables in each procedure in which you use them. For user-defined stack columns, you accomplish this by simply reissuing the original COMPUTE command in each procedure to which you are passing the stack. (You only need to specify the variable's format; do not specify its value, which is passed with the stack.)

- **Within the Master File.** You can define a virtual field in a Master File by using the DEFINE attribute. The field is then available in every procedure that accesses the data source. The virtual field is treated as part of the data source segment in which it is defined, and Maintain automatically creates a corresponding column for it—a virtual column—in every stack that references that segment.

Unlike other kinds of stack columns, you cannot update a virtual column or field, and you cannot test it in a WHERE phrase.

Example **Creating a User-defined Column**

Consider a database stack named Pay that contains information from the Employee data source. If you want to create a user-defined column named Bonus and set its value to 10 percent of each employee's current salary, you could issue the COMPUTE command to create the new column, and then issue another COMPUTE to derive the value. You place the second COMPUTE within a REPEAT loop to execute it once for each row in the stack:

```
COMPUTE Pay.Bonus/D10.2;
REPEAT Pay.FocCount Row/I4=1;
  COMPUTE Pay(Row).Bonus = Pay(Row).Curr_Sal * .10;
ENDREPEAT Row=Row+1;
```

Copying Data Into and Out of a Database Stack

You can copy data into and out of a database stack in the following ways:

- **Between a stack and a data source.** You can copy data from a data source into a stack using the NEXT and MATCH commands. You can copy data in the opposite direction, from a stack into a data source, using the INCLUDE, UPDATE, and REVISE commands. In addition, the DELETE command, while not actually copying a stack's data, reads a stack to determine which records to remove from a data source. For more information about these commands, see Chapter 7, *Command Reference*.
- **Between a stack and a Winform.** You can copy data from a stack into a Winform, and from a Winform into a stack, by specifying the stack as the source or destination of the data displayed by the Winform. This technique makes it easy for an application user to enter and edit stack data at a terminal. For more information about using stacks with Winforms, see Chapter 5, *Using the Winform Painter*.
- **From a transaction file to a stack.** You can copy data from a transaction file to a stack using the NEXT command. For more information about this command, see NEXT in Chapter 7, *Command Reference*.
- **Between two stacks.** You can copy data from one stack to another using the COPY and COMPUTE commands. For more information about these commands, see Chapter 7, *Command Reference*.

You can use any of these commands to copy data by employing the command's INTO and FROM phrases. FROM specifies the command's data source (the source stack), and INTO specifies the command's data destination (the destination stack).

Example Copying Data Between a Database Stack and a Data Source

In this NEXT command,

```
FOR ALL NEXT CustID INTO CustStack;
```

the INTO phrase copies the data (the CustID field and all of the other fields in that segment) into CustStack. The following UPDATE command,

```
FOR ALL UPDATE ExpDate FROM CustStack;
```

uses the data from CustStack to update records in the data source.

Referring to Specific Stack Rows Using an Index

Each stack has an index that enables you to refer to specific rows. Consider the earlier example in which we created the CustNames stack by issuing a NEXT command to retrieve records from the VideoTrk data source:

```
FOR ALL NEXT CustID LastName INTO CustNames
WHERE ExpDate GT 920624;
```

The first record retrieved from VideoTrk becomes the first row in the database stack, the second record becomes the second row, and so on.

	CustID	LastName	...	Zip
1	0925	CRUZ	...	61601
2	1118	WILSON	...	61601
3	1423	MONROE	...	61601
4	2282	MONROE	...	61601
5	4862	SPIVEY	...	61601
6	8771	GARCIA	...	61601
7	8783	GREEN	...	61601
8	9022	CHANG	...	61601

You can refer to a row in the stack by using a subscript. The following example refers to the third row, in which CustID is 1423:

```
CustNames (3)
```

You can use any integer value as a subscript; an integer literal (such as 3), an integer field (such as TransCode), or an expression that resolves to an integer (such as TransCode + 2).

You can even refer to a specific column in a row (that is, to a specific stack cell) by using the stack name as a qualifier:

```
CustNames (3) .LastName
```

If you omit the row subscript, the position defaults to the first row. For example,

```
CustNames .LastName
```

is equivalent to

```
CustNames (1) .LastName
```

Maintain provides two system variables associated with each stack. These variables help you manipulate single rows and ranges of rows:

- **FocCount.** This variable's value is always the number of rows currently in the stack and is set automatically by Maintain. This is very helpful when you loop through a stack, as described in the following section, *Looping Through a Stack* on page 2-14. FocCount is also helpful for checking to see if a stack is empty:

```
IF CustNames.FocCount EQ 0 THEN PERFORM NoData;
```

- **FocIndex.** This variable points to the current row of the stack. When a stack is displayed in a Winform, the Winform sets FocIndex to the row currently selected in the grid or browser. Outside of a Winform, the developer sets the value of FocIndex. By changing its value, you can point to any row you wish. For example, in one Maintain function you can increment FocIndex for the Rental stack:

```
IF Rental.FocIndex LT Rental.FocCount
  THEN COMPUTE Rental.FocIndex = Rental.FocIndex + 1;
```

You can then invoke a second function that uses FocIndex to retrieve desired records into the MovieList stack:

```
FOR ALL NEXT CustID MovieCode INTO MovieList
  WHERE VideoTrk.CustID EQ Rental (Rental.FocIndex) .CustID;
```

The syntax

```
stackname (stackname.FocIndex)
```

is identical to

```
stackname ()
```

so you can code the previous WHERE phrase more simply as follows:

```
WHERE VideoTrk.CustID EQ Rental () .CustID
```

Looping Through a Stack

The REPEAT command enables you to loop through a stack. You can control the process in different ways, so that you can loop according to several factors:

- The number of times specified by a literal or by the value of a field or expression.
- The number of rows in a stack, by specifying the stack's FocCount variable.
- While an expression is true.
- Until an expression is true.
- Until the logic within the loop determines that the loop should be exited.

You can also increment counters as part of the loop.

Example Using REPEAT to Loop Through a Stack

For example, the following REPEAT command loops through the Pay stack once for each row in the stack and increments the temporary field Row by one for each loop:

```
REPEAT Pay.FocCount Row/I4=1;  
  COMPUTE Pay(Row).NewSal = Pay(Row).Curr_Sal * 1.10;  
ENDREPEAT Row=Row+1;
```

Sorting a Stack

You can sort a stack's rows using the STACK SORT command. You can sort the stack by one or more of its columns and sort each column in ascending or descending order. For example, the following STACK SORT command sorts the CustNames stack by the LastName column in ascending order (the default order):

```
STACK SORT CustNames BY LastName;
```

Editing Stack Values

There are two ways in which you can edit a stack's values:

- **Winforms.** You can display a stack in a grid and edit its fields directly on the screen.
- **COMPUTE command.** You can use the COMPUTE command, or a simple assignment statement, to assign a value to any of a stack's cells. For example, the following command assigns the value 35,000 to the cell at the intersection of row 7 and column NewSal in the Pay stack:

```
COMPUTE Pay(7).NewSal = 35000;
```

It is important to note that if you do not specify a row when you assign values to a stack, Maintain defaults to the first row. Thus if the Pay stack has 15 rows, and you issue the following command,

```
COMPUTE Pay.NewSal = 28000;
```

the first row receives the value 28000. If you issue this NEXT command,

```
FOR 6 NEXT NewSal INTO Pay;
```

the current row of Pay defaults to one, so the six new values are written to rows one to six of Pay, and any values originally in the first six rows of Pay are overwritten. If you wish to append the new values to Pay—that is, to add them as new rows 16 through 21—you would issue this NEXT command, which specifies the starting row:

```
FOR 6 NEXT NewSal INTO Pay(16);
```

If you want to discard the original contents of Pay and substitute the new values, it is best to clear the stack before writing to it using the following command:

```
STACK CLEAR Pay;
FOR 6 NEXT NewSal INTO Pay;
```

The Default Database Stack: The Current Area

For most fields and variables referenced by a Maintain procedure, Maintain creates a corresponding column in the default database stack known as the Current Area.

The Current Area has only one row. The row includes values for the following:

- **User-defined fields.** Each user-defined field—that is, each scalar (non-stack) variable created with the COMPUTE command—exists exclusively as a column in the Current Area.
- **Data source fields.** Each data source field—that is, each field described in a Master File accessed by the Maintain procedure—has a corresponding column in the Current Area. When a data source command assigns a value—to a field, using INCLUDE or UPDATE or from a field to a stack, using NEXT or MATCH—the same value is assigned to the corresponding column in the single row of the Current Area. The final value in the Current Area is the last value written.

Developer's Tip: Using stacks is a superior way to access and manipulate data source values; stacks function more intuitively than the Current Area. Using stacks instead of the Current Area to work with data source values is recommended.

For example, if you write 15 values for NewSal to the Pay stack, the values will also be written to the NewSal column in the Current Area; since the Current Area has only one row, its value will be the fifteenth (that is, the last) value written to the Pay stack.

The Current Area is the default stack for all FROM and INTO phrases in Maintain commands. If you do not specify a FROM stack, the values come from the single row in the Current Area; if you do not specify an INTO stack, the values are written to the single row of the Current Area, so that only the last value written remains.

The standard way of referring to a stack column is by qualifying it with the stack name and a period:

stackname.columnname

Because the Current Area is the default stack, you can explicitly refer to its columns without the stack name, by prefixing the column name with a period:

.columnname

Within the context of a WHERE phrase, an unqualified name refers to a data source field (in a NEXT command) or a stack column (in a COPY command). To refer to a Current Area column in a WHERE phrase, you should refer to it explicitly by qualifying it with a period. Outside of a WHERE phrase it is not necessary to prefix the name of a Current Area column with a period, as unqualified field names will default to the corresponding column in the Current Area.

For example, the following NEXT command compares Emp_ID values taken from the Employee data source with the Emp_ID value in the Current Area:

```
FOR ALL NEXT Emp_ID Pay_Date Ded_Code INTO PayStack
WHERE Employee.Emp_ID EQ .Emp_ID;
```

If the Current Area contains columns for fields with the same field name, but that are located in different segments or data sources, you can distinguish between the columns by qualifying each with the name of the Master File and/or segment in which the field is located:

```
file_description_name.segment_name.column_name
```

If the Current Area contains columns for a user-defined field and a data source field that have the same name, you can qualify the name of the data source field column with its Master File and/or segment name; an unqualified reference will refer to the user-defined field column.

Maximizing Database Stack Performance

When you use database stacks, there are several things you can do to optimize performance:

- **Filter out unnecessary rows.** When you read records into a stack, you can prevent the stack from growing unnecessarily large by using the WHERE phrase to filter out unneeded rows.
- **Clear stacks when done with data.** Maintain automatically releases a stack's memory at the end of a procedure, but if in the middle of a procedure you no longer need the data stored in a stack, you can clear it immediately by issuing the STACK CLEAR command, freeing its memory for use elsewhere.
- **Do not reuse a stack for an unrelated purpose.** When you specify a stack as a data source or destination in certain contexts (in the NEXT, MATCH, and INFER commands, and in the Winform Painter for controls), you define the columns that the stack will contain. If you use the same stack for two unrelated purposes, it will be created with the columns needed for both, making it unnecessarily wide.

Controlling a Procedure's Flow

Maintain provides many different ways of controlling a procedure's flow of execution. You can:

- **Nest** a block of code. In commands in which you can nest another command—such as in an IF command—you can nest an entire block of commands in place of a single one by defining the block using the BEGIN command. BEGIN is described in *BEGIN* in Chapter 7, *Command Reference*.
- **Loop** through a block of code a set number of times, while a condition remains true or until it becomes true, using the REPEAT command. REPEAT is described in *REPEAT* in Chapter 7, *Command Reference*.
- **Branch unconditionally** to a block of code called a Maintain function (“Maintain functions” are often referred to more briefly as functions, and are also known as cases). You define the function using the CASE command, and can invoke it in a variety of ways. When the function terminates, it returns control to the command following the function invocation. The CASE command is described in *CASE* in Chapter 7, *Command Reference*.

Alternatively, you can branch to a function, but not return upon termination, by invoking the function using the GOTO command. GOTO is described in *GOTO* in Chapter 7, *Command Reference*.

- **Branch conditionally** using the IF command. If the expression you specify in the IF command is true, the command executes a PERFORM or GOTO command nested in the THEN phrase, which branches to a function. IF is described in *IF* in Chapter 7, *Command Reference*.

Alternatively, you can nest a different command, such as a BEGIN command defining a block of code, to be conditionally executed by the IF command.

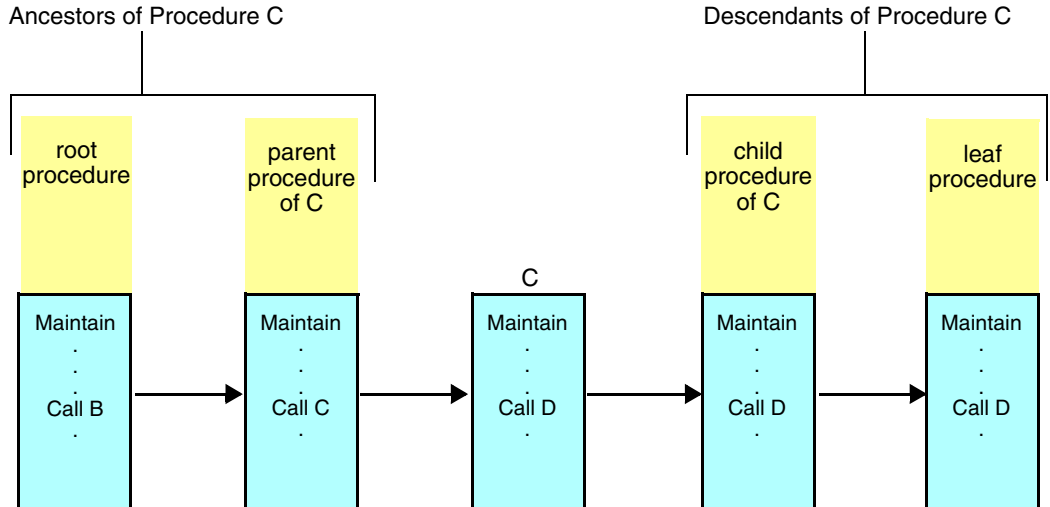
- **Trigger** a function or system action in response to a user event. When a user performs the event in a Winform at run time, the event triggers the function or system action that you have specified. Event-driven processing is described in *Winforms and Event-driven Processing* on page 2-25.

Executing Other Maintain Procedures

You can call one Maintain procedure from another with the CALL command. (*Maintain procedure* here means any procedure of Maintain language commands.) CALL simplifies the process of modularizing an application; software designed as a group of related modules tends to be easier to debug, easier to maintain, and lends itself to being reused by other applications—all of which increase your productivity.

CALL also makes it easy to partition an application, deploying each type of logic on the platform on which it will run most effectively.

The following diagram illustrates how to describe the relationship between called and calling procedures. It describes a sequence of five procedures from the perspective of the middle procedure, C FOCEXEC. (Note that a root procedure is also called the starting procedure.)



Example **Calling Another Maintain Procedure**

Consider the EMPUPDAT procedure:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO EmpStack;
.
.
.
CALL NewClass;
.
.
.
END
```

This calls the NEWCLASS procedure:

```
MAINTAIN
.
.
.
END
```

In this example, EMPUPDAT is the parent procedure and NEWCLASS is the child procedure. When the child procedure—and any procedures that it, in turn, has invoked—has finished running, control returns to the parent.

Passing Variables Between Procedures

All user variables (both stacks and simple, or scalar, variables) are local to a procedure, not global to the application. In other words, to protect them from unintended changes in other parts of an application, you cannot directly refer to a variable outside of the procedure in which it is found (with the exception of the FocError, FocErrorRow, FocFetch, and FocCurrent transaction variables). However, you can access a variable's data in other procedures, simply by passing it as an argument from one procedure to another.

To pass variables as arguments, you only need to name them in the CALL command, and then again in the corresponding MAINTAIN command. Some variable attributes must match in the CALL and MAINTAIN commands:

- **Sequence.** The order in which you name stacks and simple variables must be identical in the CALL and corresponding MAINTAIN commands.
- **Data type.** Stack columns and simple variables must have the same data type (for example, integer) in both the parent and child procedures.
- **Stack column names.** The names of stack columns do need to match; if a column has different names in the parent and child procedures, it is not passed.

Other attributes need not match:

- **Stack and scalar variable names.** The names of stacks and simple variables specified in the two commands do not need to match.
- **Other attributes.** All other data attributes, such as length and precision, do not need to match.
- **Simple variables.** If you pass an individual stack cell, you must receive it as a simple Current Area variable.

Once you have passed a variable to a child procedure, you must define it in that procedure. How you define it depends upon the type of variable:

- **User-defined columns and fields.** You must redefine each user-defined variable using a COMPUTE command. You only need to specify the variable's format, not its value. For example, the following COMPUTE command redefines the Counter field and the FullName column:

```
COMPUTE Counter/A20;
        EmpStack.FullName/A15;
```

- **Data source and virtual stack columns.** You can define a stack's data source columns and virtual columns in one of two ways: implicitly, by referring to the stack columns in a data source command, or explicitly, by referring to them using the INFER command. For example:

```
INFER Emp_ID Pay_Date INTO EmpStack;
```

The INFER command declares data source fields and the stack with which they are associated. You can specify one field for each segment you want in the stack or simply one field each from the anchor and target segments of a path you want in the stack.

While INFER reestablishes the stack's definition, it does not retrieve any records from the data source.

Once a variable has been defined in the child procedure, its data becomes available. If you refer to stack cells that were not assigned values in the parent procedure, they are assigned default values (such as spaces or zeros) in the child procedure, and Maintain displays a message warning that they have not been explicitly assigned any values.

When the child procedure returns control back to the parent procedure, the values of stacks and simple variables specified as INTO variables are passed back to the parent. The values of stacks and simple variables specified only as FROM variables are not passed back.

Example **Passing Data Between Maintain Procedures**

For example, this procedure,

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO EmpStack;
.
.
.
CALL NewClass FROM EmpStack CourseStack INTO CourseStack;
.
.
.
END
```

calls the NEWCLASS procedure:

```
MAINTAIN FROM StudentStack CourseStack INTO CourseStack
.
.
.
END
```

EmpStack and CourseStack in the parent procedure correspond to StudentStack and CourseStack in the child procedure.

Accessing Data Sources in the Child Procedure

If a child procedure accesses a data source, whether retrieving or writing records, you must specify the data source in the MAINTAIN command. This is done the same way as for a stand-alone procedure. For example:

```
MAINTAIN FILES Employee AND EducFile FROM StuStk INTO CoursStk
.
.
.
END
```

Data Source Position

Each Maintain procedure tracks its own position in the data source. When you first call a procedure, Maintain positions you at the beginning of each segment in each data source accessed within that procedure; after navigating through a data source, you can reposition to the beginning of a segment by issuing the REPOSITION command. Each procedure's data source positions are independent of the positions established in other procedures.

When a child procedure returns control to its parent, by default it clears its data source positions. You can specify that it retain its positions for future calls by using the KEEP option, as described in the *Optimizing Performance: Data Continuity and Memory Management* on page 2-23.

Optimizing Performance: Data Continuity and Memory Management

By default, when you terminate a child procedure, Maintain clears its data from memory to save space. You can optimize your application's performance by specifying, each time you terminate a child procedure, how you want Maintain to handle the procedure's data. You have two options, based on how often you will call a given procedure over the course of an application:

- If you will call the procedure frequently, use the KEEP option to make the procedure run faster by retaining its data between calls.

This option provides data continuity; the procedure's data carries over from the end of one invocation to the beginning of the next. That is, the next time you call the procedure, its variables and data source position pointers start out with the same values that they held when the procedure was last terminated. You can use these values or, if you wish, re-initialize them using the DECLARE (or COMPUTE) and REPOSITION commands.

Variables passed by the parent procedure are not affected by data continuity, since the child procedure receives them directly from the parent procedure at the beginning of each call.

The KEEP option does not issue an implied COMMIT command at the end of a child procedure. When a child procedure with an open logical transaction returns to its parent procedure and specifies KEEP, the transaction continues into the parent.

- If you will call the procedure rarely, use the RESET option to reduce memory consumption by freeing the procedure's data at the end of each call.

This option does not provide data continuity; all of the procedure's variables and data source position pointers are automatically initialized at the beginning of each procedure.

The RESET option issues an implied COMMIT command at the end of a child procedure. When a child procedure with an open logical transaction returns to its parent procedure using RESET, the transaction is closed at the end of the child procedure.

For more information about transactions spanning procedures, see *Designing Transactions That Span Procedures* on page 2-39.

You can specify how a procedure will handle its data in memory by terminating it with the GOTO END command qualified with the appropriate memory-management phrase. The syntax is

```
GOTO END [KEEP|RESET] ;
```

where:

KEEP

Terminates the procedure, but keeps its data—the values of its variables and data source position pointers—in memory. It remains in memory through the next invocation, or (if it is not called again) until the application terminates. The procedure does not issue an implied COMMIT command to close an open logical transaction.

RESET

Terminates the procedure, clears its data from memory, and issues an implied COMMIT command to close an open logical transaction. This is the default.

You can use both options in the same procedure. For example, when you are ready to end a child procedure, you could evaluate what logic the procedure will need to perform when it is next called and then branch accordingly either to keep data in memory, saving time and providing data continuity, or to clear data from memory to conserve space.

Winforms and Event-driven Processing

Maintain introduces a new graphical environment for displaying and editing data, and a new paradigm for developing applications. At the heart of both of these features are Winforms. This is a sample Winform:

The screenshot shows a window titled "Customer List". At the top left, it says "Page 1 of 36". The main content area displays the following information:

Customer ID: 0925
 Name: IUY CRUZ
 Address: 86 ELLIOTT AVE.
 PEORIA IL 61601
 Phone: 3095550925

Below this information is a table with a scroll bar on the right side:

	MOVIECODE	COPY	RETURDATE	FEE
1	001MCA	1	91/06/29	2.00
2	692PAR	2	91/06/29	2.00

At the bottom of the window, there are four buttons: "Prev F7", "Next F8", "Quit F3", and "Done F4".

Winforms are windows that you can use to display, enter, and edit data. You can also use Winforms to present choices to users and to control the flow of your application. For those familiar with MODIFY, this is similar to the role that CRTFORMs and Dialogue Manager Windows play in a MODIFY application, but Winforms go far beyond them. Winforms offer the following features to the Maintain developer:

- **Full graphical environment.** Winforms provide a full graphical environment, complete with drop-and-click menus, multiple button types, and scroll bars.
- **Flexibility and features.** Winforms combine the field-oriented features of CRTFORMs with the control-oriented features of Dialogue Manager Windows, and introduce additional features designed for stacks and functions. One user interface now provides you with multiple solutions.

- **Event-driven processing.** Winforms are more responsive to the needs of a user because they recognize many types of user activity on the screen—that is, many types of screen events. A Winform recognizes what the user does on the screen with a cursor and function keys; for example, it knows when a user selects an area on the screen that has been defined as a button.

Winforms also enable you to define triggers, which are links between these events and the application's functions: each time a specified event occurs, Maintain automatically invokes the corresponding function or system action. (Functions are also known as cases.) If you use events to trigger the application's functions, you can give the user more freedom—for example, over which editing tasks to perform, and in which order. You can also give the user access to more functionality, and more types of data, on a single screen. Event-driven processing gives the user more flexibility over the application, even as it gives the application more control over the user interface.

- **Event-driven development.** Maintain provides you with a simple way of developing event-driven applications: event-driven development. Because much of an application's flow can be controlled from Winforms, you can develop the application as you paint the Winform: first design the visual layout, then define triggers, and finally code the trigger functions—all from within the Winform Painter.

The following sections introduce you to using Winforms and creating event-driven applications. Winforms, and the Winform Painter used to design and create them, are described in greater detail in Chapter 5, *Using the Winform Painter*.

How to Use Winforms

Winforms are Graphical User Interface (GUI) windows, similar to the windows used in Microsoft Windows. Winforms have standard window features, such as:

- **A title bar** that identifies the Winform.
- **Scroll bars** that enable you to move a grid's contents vertically and horizontally if they extend beyond the grid's border.
- **A control box**, in the upper-left corner, that enables you to perform basic functions such as moving and resizing the Winform.

You can display multiple Winforms on the screen. One Winform can be active at a time—for example, to receive data from the keyboard—and you can transfer control from one Winform to another.

Designing a Winform

When you design a Winform, you can set a number of properties, including:

- **Pop-up.** Do you want the Winform to disappear automatically once it becomes inactive?
- **Border.** Do you want the Winform to have a visible border?
- **Colors.** If the Winform is to be displayed on monitors that support color, what color do you wish individual fields, the Winform's background, and other screen elements to be?

Winforms offer a diverse set of ways by which an end user can select options, invoke procedures, display and edit fields, and get helpful information. For example, if you want the user to select an option or procedure, you can use any of the following controls (controls are referred to as objects in the Winform Painter):

- **Command buttons.** You can specify a function or special function to be performed when the user presses a button. Common examples are OK and Cancel buttons.
- **Radio buttons.** The user can select one of a mutually exclusive group of options. For example, an applicant could identify his or her gender.
- **List and combo boxes.** The user can select an option from a dynamic list of choices.
- **Check boxes.** The user can select an option. For example, an administrator could indicate that a student is paying by credit card.

If you wish to display or edit data, you can do so for:

- **Individual fields.** You can take the field's initial value from a stack or from the Current Area. If you specify a stack, you can move through the field values one row at a time, using the browser.
- **An entire stack.** You can display a stack in a grid. If the stack is larger than the grid, you can move through it using scroll bars.
- **Selected stack columns.** If you need only limited information from the stack, you can limit the grid to selected stack columns.

You can provide the end user with helpful information by:

- **Including explanatory text** on the screen.
- **Drawing a frame** around related items on the screen.
- **Providing online help** in the form of pop-up Winforms containing useful information. You can invoke the help Winforms using a trigger, such as the PF1 key.

Using Winforms in Block Mode

Mainframe monitors—whether they are actual terminals or personal computers emulating terminals—communicate with the host computer in block mode. Each time the user presses Enter or a function key, all of the information on the screen is sent, as a single block, to the host for processing. For example, if the user tabs to a new position on the screen and then types data into three fields, the host does not become aware of these actions until the user presses *Enter* or a function key.

This has important implications for using Winforms: to perform any action that requires selecting an item on the screen, you must position the cursor on the item—such as a pull-down menu—and then press Enter or a function key.

To perform an action that is triggered by a function key, simply press the key. If the action depends upon the screen context, first position the cursor on the appropriate item, and then press the function key. For example, in the Winform Painter the PF5 key moves a control. If the cursor is in an empty spot on the screen, pressing PF5 does not accomplish anything. If the cursor is on a grid, PF5 moves the grid; if the cursor is on a field, PF5 moves the field.

Adjusting Winforms for Your Terminal Configuration

Different terminals and terminal emulators process screen information in different ways. Some configurations support dashed lines instead of solid Winform borders. For solid borders, SBORDER must be set to ON in your FOCUS session. (See the *Developing Applications* manual for information about SET commands.)

If a Winform's visual elements—such as check boxes and vertical scroll bars—are displayed incorrectly on the screen, you can adjust the Winform facility's terminal emulator setting. To do so for:

- Terminals on which applications will be deployed, issue the following command at the FOCUS command prompt:
`EX MSETUP`
- The Winform Painter when developing applications, select *Terminal* from the Preferences option of the Winform Painter's File menu.

For more information, see *Terminal* in Chapter 5, *Using the Winform Painter*.

Optimizing Winform Performance

You can optimize Winform performance by deploying all Winforms in a single Maintain procedure. This technique reduces memory consumption, and applies mostly to CMS.

Designing Event-driven Applications

The flow of control in conventional processing is mostly pre-determined—that is, the programmer determines the few paths which the user will be able to take through the procedure.

To make your application more responsive to the user—using the graphical user interface—Maintain provides triggers and event-driven processing. A trigger is an action that is invoked—triggered—by a specified event. Each time that the event occurs, the trigger action (also known as an event handler) is invoked. In Maintain, the trigger action is a function or system action, and the event is something the user does in a Winform. For example, you might create a data retrieval trigger that notices whenever a user clicks a certain button on a Winform, and reacts by invoking a function that reads a data source and displays the data in the Winform.

Creating Event-driven Applications

Developing a request by writing out sequential lines of code may be sufficient for conventional linear processing, but event-driven processing demands event-driven development. Developing an application in this way enables you to build much of the application's logic around the user interface. In effect, you paint the application as you paint the interface in the Winform Painter. For example, you could start by developing a Winform, then assigning a trigger to a particular Winform event, specifying the action (that is, the function) associated with the trigger, and finally coding the function.

Reading From a Data Source

Most applications need to read data from a data source. Before reading, you first must select the record in which the data resides. There are five ways of selecting records:

- **By field value for an entire set of records.** Use the NEXT command. The WHERE phrase enables you to select by value, and the FOR ALL phrase selects the entire set of records that satisfy the WHERE selection condition. The basic syntax for this is:

```
FOR ALL NEXT fields INTO stack WHERE selection_condition;
```

- **By field value for a sequence (subset) of records.** Use the NEXT command. This is similar to the technique for a set of records, except that it employs the FOR *n* phrase, selecting—at the current position in the data source—the first *n* records that satisfy the WHERE condition. The basic syntax for this is:

```
FOR n NEXT fields INTO stack WHERE selection_condition;
```

- **By field value, one segment at a time, one record at a time.** Use the MATCH command. The basic syntax for this is:

```
MATCH fields [FROM stack] [INTO stack];
```

- **Sequentially for a sequence (subset) of records.** Use the NEXT command. This technique employs the FOR *n* phrase to select the next *n* records. The basic syntax for this is:

```
FOR n NEXT fields INTO stack;
```

- **Sequentially, one segment instance at a time, one record at a time.** Use the NEXT command. The basic syntax for this is:

```
NEXT fields [INTO stack];
```

You can read from individual data sources, and from those that have been joined. Maintain supports joins that are defined in the Master File. For information about defining joins in the Master File, see the FOCUS Interface documentation for the types of data sources that you wish to join, and the *Describing Data* manual for FOCUS data sources.

You can evaluate the success of a command that reads from a data source by testing the FocError system variable, as described in *Evaluating the Success of a Simple Data Source Command* on page 2-32.

The NEXT and MATCH commands are described in detail in Chapter 7, *Command Reference*.

Repositioning Your Location in a Data Source

Each time you issue a NEXT command, Maintain begins searching for records from the current position in the data source. For example, if your first data source operation retrieved a set of records,

```
FOR ALL NEXT CustID INTO SmokeStack
  WHERE ProdName EQ 'VCR DUST COVER';
```

then Maintain will have searched sequentially through the entire data source, so the current position marker will now point to the end of the data source. If you then issue another NEXT command,

```
FOR ALL NEXT LastName FirstName INTO CandyStack
  WHERE ProdName EQ 'CANDY';
```

Maintain searches from the current position to the end of the data source; since the current position *is* the end of the data source, no records are retrieved.

When you want a NEXT command to search through the entire data source (as is often the case when you wish to retrieve a set of records) you should first issue the REPOSITION command to move the current position marker to the beginning of the data source.

Example Repositioning to the Beginning of the Data Source

The following REPOSITION command specifies the CustID field in the root segment, and so moves the current position marker for the root segment chain and all of its descendant chains back to the beginning of the chain (in effect, back to the beginning of the data source):

```
REPOSITION CustID;
FOR ALL NEXT LastName FirstName INTO CandyStack
  WHERE ProdName EQ 'CANDY';
```

Writing to a Data Source

Writing to a data source is the heart of transaction processing applications. Maintain provides the following commands to write to a data source:

- **INCLUDE**, which adds the specified new segment instances to a data source.
- **UPDATE**, which updates the specified fields in a data source.
- **REVISE**, which adds new segment instances and updates the specified fields in existing segment instances.
- **DELETE**, which removes the specified segment instances from a data source.

These commands are described in detail in Chapter 7, *Command Reference*.

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

Evaluating the Success of a Simple Data Source Command

When you issue a command that reads or writes to a data source, you should determine whether the command was successful. A data source command might not be successful if you attempt to insert a record that already exists; if you attempt to update, delete, or read a record that does not exist; or if the command is interrupted by a system failure.

When you issue a command that reads or writes to a data source, if the command is:

- **Successful**, Maintain automatically sets the transaction variable FocError to 0 (zero), and writes to the data source.
- **Unsuccessful**, Maintain sets FocError to a non-zero value, and does not write to the data source.

Example Evaluating the Success of an UPDATE Command

The following function updates the VideoTrk data source for new video rentals. If the UPDATE command is unsuccessful, the application invokes a function that displays a message to the user. The line that evaluates the success of the command is shown in bold:

```
CASE UpdateCustOrder
  UPDATE ReturnDate Fee FROM RentalStack;
IF FocError NE 0 THEN PERFORM ErrorMessage;
ENDCASE
```

Evaluating the Success of a Stack-based Write Command

When you write a set of stack rows to a data source, if you specify more rows than there are matching data source records, this does not invalidate the write operation: Maintain attempts to write all the matching rows to the data source. For example, the following UPDATE command specifies 15 rows, but there are only 12 matching rows; all 12 are written to the data source.

```
FOR 15 UPDATE Curr_Sal FROM NewSalaries;
```

When you write a set of stack rows to a data source, if one row fails:

- The rows preceding the failed row are written to the data source.
- The rows following the failed row are ignored.
- FocError is set to a non-zero value, signaling an error.
- FocErrorRow is set to the number of the failed row.

Data source logic errors include attempting to insert an existing record, or to update or delete a nonexistent record.

To determine whether an entire stack was successfully written to the data source, test FocError immediately following the data source command. If FocError is not 0, you can determine which row caused the problem by testing FocErrorRow; you can then reprocess that row.

If you do not wish to take advantage of this flexibility, and instead prefer to invalidate *all* the rows of the stack if any of them are unsuccessful, you can bracket the data source command in a logical transaction that you can then roll back. Logical transactions are discussed in *Transaction Processing* on page 2-34.

Row failure when committing to a data source. If a stack-based write command is part of a logical transaction, and the write command succeeds when it is issued but fails when the application tries to commit the transaction, Maintain rolls back all of the write command's rows, along with the rest of the transaction. (For example, a write command might fail at commit time because another user has already changed one of the records to which the command is writing.) Transaction processing is described in *Transaction Processing* on page 2-34.

Example Evaluating a Stack-based Update Command

The NewSalaries stack has 45 rows. The following command updates the Employee data source for all the rows in NewSalaries:

```
FOR ALL UPDATE Curr_Sal FROM NewSalaries;
```

If there is no data source record that matches the thirtieth row, Maintain updates the records matching the first 29 rows and ignores rows 30 and higher.

Transaction Processing

You are familiar with individual data source operations that insert, update, or delete data source segment instances. However, most applications are concerned with “real-world” transactions, like transferring funds or fulfilling a sales order, that require several data source operations. These data source operations may access several data sources, and may be issued from several procedures. We call such a collection of data source operations a *logical transaction*, also known as a logical unit of work.

This and related topics describe how Maintain ensures transaction integrity at the application level. At the data source level, each database management system (DBMS) implements transaction integrity in its own way; see your DBMS vendor’s documentation for DBMS-specific information. For FOCUS data sources, this DBMS-specific information is presented in *Ensuring Transaction Integrity for FOCUS Data Sources* on page 2-44. For DB2, you can find some suggested strategies for writing Maintain transactions to DB2 data sources in *Ensuring Transaction Integrity for DB2 Data Sources* on page 2-50. For many other types of data sources, you can also apply the strategies described in *Ensuring Transaction Integrity for DB2 Data Sources* on page 2-50, changing DBMS-specific details when necessary.

Example Describing a Transfer of Funds as a Logical Transaction

A banking application would define a transfer of funds from one account to another as one logical transaction comprising two update operations:

- Subtracting the funds from the source account (UPDATE Savings FROM SourceAccts).
- Adding the funds to the target account (UPDATE Checking FROM TargetAccts).

Procedure How to Process a Logical Transaction

To process a logical transaction, follow these steps:

1. **DBMS requirements.** Your data sources' database management system (DBMS) may require that you perform some tasks to enable transaction integrity; see your database vendor's documentation for information. You can set some native DBMS parameters through FOCUS; for more information, see the FOCUS Interface manual for your DBMS.

For FOCUS data sources, you must set the COMMIT environment variable to ON, and to issue a USE command to specify which FOCUS Database Server will manage concurrent access to the data source. For more information, see *Ensuring Transaction Integrity for FOCUS Data Sources* on page 2-44.

2. **Develop the transaction logic.** Code the data source commands and related logic that read from the data sources, write to the data sources, and evaluate the success of each data source command.
3. **Define the transaction boundary.** Code a COMMIT command, and any other supporting commands, to define the transaction's boundary. For more information, see *Defining a Transaction* on page 2-36.
4. **Evaluate the transaction's success.** Test the FocCurrent transaction variable to determine whether the transaction was successfully written to the data source, and then branch accordingly. For more information, see *Determining Whether a Transaction Was Successful* on page 2-41.

Why Is Transaction Integrity Important?

The advantage of describing a group of related data source commands as one logical transaction is that the transaction is valid and written to the data source only if all of its component commands are successful. When you attempt to commit a transaction, you are ensured that if part of the transaction fails, none of the transaction will be written to the data source. This is called transaction integrity.

When is transaction integrity important? Whenever a group of commands are related and are only meaningful within the context of the group. In other words, whenever the failure of any one command in the transaction at commit time would invalidate the entire transaction.

Transaction integrity is an all-or-nothing proposition: either all of the transaction is written to the data source when you commit it, or all of it is rolled back.

Example **Why Transaction Integrity Is Essential to a Bank**

Consider a banking application that transfers funds from a savings account to a checking account. If the application successfully subtracts the funds from the savings account, but is interrupted by a system problem before it can add the funds to the checking account, the money would “disappear,” unbalancing the bank’s accounts.

The two update commands (subtracting and adding funds) must be described as parts of a single logical transaction, so that the subtraction and addition updates are not written to the data source independently of each other.

Defining a Transaction

You define a logical transaction by issuing a COMMIT or ROLLBACK command following the transaction’s last data source command. (For simplicity, the remainder of this topic refers to COMMIT only, but unless stated otherwise, both commands are meant.) For example, the beginning of your application is the beginning of its first logical transaction. The data source commands that follow are part of the transaction. When the application issues its first COMMIT command, it marks the end of the first transaction.

The data source commands that follow the first COMMIT become part of the second logical transaction; the next COMMIT to be issued marks the end of the second transaction, and so on.

The COMMIT command defines the transaction’s boundary. All data source commands issued between two COMMIT commands are in the same transaction. (This explanation describes the simplest case, in which a transaction exists entirely within a single procedure. When a transaction spans procedures, you have several options for deciding how to define a transaction’s boundary, as described in *When an Application Ends With an Open Transaction* on page 2-41.)

Example **Defining a Simple Transfer of Funds Transaction**

For example, transferring money from a savings account to a checking account requires two update commands. If you want to define the transfer, including both updates, as one logical transaction, you could use the following function:

```
CASE TransferMoney
  UPDATE Savings FROM SourceAccts
  UPDATE Checking FROM TargetAccts
  COMMIT
ENDCASE
```

Reference **When Does a Data Source Command Cause a Transaction to Fail?**

A data source command can fail for many reasons—for example, an UPDATE command might try to write to a record that never existed because a key was mistyped, or an INCLUDE command might try to add a record that has already been added by another user.

In some cases, when a command fails, you might want to keep the transaction open and simply resolve the problem that caused the command to fail. For example, in the first case—attempting to update a record that doesn't exist—you might wish to ask the end user to correctly re-enter the customer code (which is being used as the record's key). In other cases, you might wish to roll back the entire transaction.

If a data source command fails, it will only cause the logical transaction that contains it to be automatically rolled back in certain circumstances. The deciding factor is *when* a data source command fails. If a data source command fails when the transaction:

- **Is open** (that is, when the application issues the data source command), the transaction remains open, and the failed data source command does not become part of the transaction. This means that, if the application later attempts to commit the transaction, because the failed data source command is not part of the transaction, it will not affect the transaction's success or failure.

You can evaluate the success of a data source command in an open transaction by testing the value of the FocError system variable immediately after issuing the command. If you wish the failure of the data source command to roll back the transaction, you can issue a ROLLBACK command.

- **Is being closed** (that is, when the application tries to commit the transaction), the failure of the data source command to be written to the data source causes the transaction to fail, and the entire transaction is automatically rolled back.

Canceling a Transaction

A transaction that is ongoing and has not yet been committed is called an open transaction. If you ever need to cancel an open transaction, you can do so by issuing a ROLLBACK command. ROLLBACK voids any of the transaction's data source commands that have already been issued, so that none of them are written to the data source.

Transactions and Data Source Position

When a logical transaction is committed or rolled back, it resets all position markers in all the data sources that are accessed by the transaction's procedures. (Resetting a data source's position markers points them to the beginning of the data source's segment chains.)

How Large Should a Transaction Be?

A transaction is at its optimal size when it includes only those data source commands that are mutually dependent upon each other for validity. If you include "independent" commands in the transaction and one of the independent commands fails when you try to commit the transaction, the dependent group of commands will be needlessly rolled back.

For example, in the following banking transaction that transfers funds from a savings account to a checking account,

```
CASE TransferMoney
  UPDATE Savings FROM SourceAccts
  UPDATE Checking FROM TargetAccts
  COMMIT
ENDCASE
```

you should not add an INCLUDE command to create a new account, since the validity of transferring money from one account to another does not depend upon creating a new account.

Another reason for not extending transactions unnecessarily is that, in a multi-user environment, the longer a transaction takes, the more likely it is to compete for records with transactions submitted by other users. Transaction processing in a multi-user environment is described in *Concurrent Transaction Processing* on page 2-42.

Designing Transactions That Span Procedures

Logical transactions can span multiple Maintain procedures. If a Maintain procedure with an open transaction passes control to a non-Maintain procedure (for example, a report procedure), the open transaction is suspended; when control next passes to a Maintain procedure, the transaction picks up from where it had left off.

When a transaction spans several procedures, you will usually find it easier to define the transaction's boundaries if you commit it in the highest procedure in the transaction (that is, in the procedure closest to the root procedure). Committing a transaction in a descendant procedure of a complex application, where it is more difficult to track the flow of execution, makes it difficult to determine the transaction's boundaries (that is, to know which data source commands are being included in the transaction).

When a child procedure returns control to its parent procedure, and the child has an open logical transaction, you have two options:

- **You can continue the child's open transaction** into the parent procedure when the child returns control to the parent. Simply specify the KEEP option when you return control with the GOTO END command.
- **You can close the child's open transaction** automatically at the end of the child procedure. By default, Maintain issues an implied COMMIT command to close the open transaction. You can also specify this behavior explicitly by coding the RESET option when you return control with the GOTO END command.

RESET and KEEP are both described in *Optimizing Performance: Data Continuity and Memory Management* on page 2-23.

Example Moving a Transaction Boundary Using GOTO END KEEP

Consider a situation where procedure A calls procedure B, and procedure B then calls procedure C. The entire application contains no COMMIT commands, so the initial logical transaction continues from the root procedure (A) through the descendant procedures (B and C). C and B both return control to their parent procedure using a GOTO END command.

The table below shows how specifying or omitting the KEEP option when procedures B and C return control affects the application’s transaction boundaries—that is, how the choice between KEEP and the implied COMMIT determines where the initial transaction ends, and how many transactions follow.

C returns to B with...	B returns to A with...	Transaction boundaries ()
KEEP	KEEP	A-B-C-B-A one transaction
KEEP	implied COMMIT	A-B-C-B A two transactions
implied COMMIT	KEEP	A-B-C B-A two transactions
implied COMMIT	implied COMMIT	A-B-C B A three transactions

Designing Transactions That Span Data Source Types

If a transaction writes to multiple types of data sources, each database management system (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. This is known as a broadcast commit.

For example, if you issue a Maintain procedure against the FOCUS data sources Employee and JobFile and a DB2 data source named Salary, the success or failure of the COMMIT against Salary is independent of its success against Employee and JobFile. It is possible for COMMIT to be successful against Salary and write that part of the transaction, while being unsuccessful against Employee and JobFile and roll back those parts of the transaction.

When an Application Ends With an Open Transaction

If an application terminates while a logical transaction is still open, Maintain issues an implied COMMIT command to close the open transaction, ensuring that any data source commands issued after the last explicit COMMIT are accounted for. (The only exception is that if your FOCUS session abnormally terminates, Maintain does not issue the implied COMMIT, and any remaining uncommitted data source commands are rolled back.)

Determining Whether a Transaction Was Successful

When you close a transaction by issuing a COMMIT or ROLLBACK command, you must determine whether the command was successful. If a COMMIT command is successful, then the transaction it closes has been successfully written to the data source; if a ROLLBACK command is successful, the transaction it closes has been successfully rolled back.

The system variable FocCurrent provides the return code of the most recently issued COMMIT or ROLLBACK command. By testing the value of FocCurrent immediately following a COMMIT or ROLLBACK command, you can determine whether the transaction was successfully committed or rolled back. If the value of FocCurrent is:

- **Zero**, the command was successful.
- **Not zero**, the command was unsuccessful.

FocCurrent is global to all the procedures in a transaction, and so does not need to be passed as an argument between procedures.

Example Evaluating the Success of a Transaction

The following function commits a transaction to a data source. If the transaction is unsuccessful, the application invokes another function that writes to a log and then begins a new transaction. The line that evaluates the success of the transaction is shown in bold:

```
CASE TransferMoney
  UPDATE AcctBalance FROM SourceAccts
  UPDATE AcctBalance FROM TargetAccts
  COMMIT
  IF FocCurrent NE 0 THEN PERFORM BadTransfer
ENDCASE
```

Concurrent Transaction Processing

Several applications or users often need to share the same data source. This sharing can lead to problems if they try to access a record concurrently—that is, if they try to process the same data source record at the same time.

To ensure the integrity of a data source, concurrent transactions must run as if they were isolated from each other; one transaction's changes to a data source must be concealed from all other transactions *until that transaction is committed*. To do otherwise runs the risk of exposing open transactions to interim inconsistent images of the data source, and consequently corrupting the data source.

To prevent users from corrupting the data in this way, the database management system must coordinate concurrent access. There are many strategies for doing this. No matter which type of data source you use, Maintain respects your DBMS's concurrency strategy and lets it coordinate access to its own data sources.

For more information about how your DBMS handles concurrent access, see your DBMS vendor's documentation. For FOCUS data sources, this information is presented in *Ensuring Transaction Integrity for FOCUS Data Sources* on page 2-44. For DB2, you can find some suggested strategies for writing Maintain transactions to DB2 data sources in *Ensuring Transaction Integrity for DB2 Data Sources* on page 2-50. For many other types of data sources, you can also apply the strategies described in *Ensuring Transaction Integrity for DB2 Data Sources* on page 2-50, changing DBMS-specific details when necessary.

Example Why Concurrent Access to a Data Source Must Be Carefully Managed

Consider the following two applications that access the Employee data source:

- The Promotion application reads a list of employees who have received promotions and updates their job codes to correspond to their new positions.
- The Salary application, run once at the beginning of each year, checks every employee's job code and gives each employee an annual raise based on his or her job title. For example, assistant managers (job code A15) will earn \$30,000 in the new year, and managers (A16) will earn \$40,000.

Joan Irving is an assistant manager. Consider what happens when these two applications try to access and update the same record at the same time, without any coordination:

1. The Promotion application reads Irving's record and, based on information in a transaction file that she has been promoted to manager, computes her new job code (A16).
2. The Salary application reads Irving's record and, based on her job code in the data source (A15), computes her new salary (\$30,000).
3. The Promotion application writes the new job code (A16) to the data source.
4. The Salary application writes the new salary (\$30,000) to the data source.

Remember the earlier business rule (assistant managers earn \$30,000, managers earn \$40,000). Because two applications accessed the same record at the same time without any coordination, the rule has been broken (Joan Irving has a manager's job code but an assistant manager's salary). The data source has become internally inconsistent.

Ensuring Transaction Integrity for FOCUS Data Sources

Each database management system (DBMS) supports transaction integrity in its own way. The FOCUS DBMS manages concurrent access to FOCUS data sources using the FOCUS Database Server, and uses certain commands to identify transaction integrity attributes. (The FOCUS Database Server was formerly known as a sink machine or the Simultaneous Usage facility on some platforms.)

To ensure transaction integrity for FOCUS data sources, perform the following tasks:

- **Set COMMIT.** Set the COMMIT environment variable to ON. This enables the COMMIT and ROLLBACK commands for FOCUS data sources, and enables the use of the FOCUS Database Server. For more information, see *Setting COMMIT* on page 2-44.
- **Select which segments will be verified for changes.** Set the PATHCHECK environment variable to specify the type of segments for which the FOCUS Database Server will verify change. This is optional: you can accept the default setting. For more information, see *Selecting Which Segments Will Be Verified for Changes* on page 2-47.
- **Identify the FOCUS Database Server.** Identify which FOCUS Database Server will manage concurrent access to each FOCUS data source. For more information, see *Identifying the FOCUS Database Server* on page 2-48.
- **Activate the FOCUS Database Server.** For more information, see the *Simultaneous Usage Reference Manual*.

Setting COMMIT

You must have set the COMMIT environment variable to ON before using the COMMIT and ROLLBACK commands for FOCUS data sources, and before using the FOCUS Database Server. It is recommended that you set COMMIT at the beginning of the Maintain application's root procedure (preceding the MAINTAIN command), and that you reset it to its initial value when the application finishes. This avoids interfering with any MODIFY applications that your site may run.

Syntax **How to Set COMMIT**

The COMMIT environment variable enables transaction integrity for FOCUS data sources. To set COMMIT, issue the SET COMMIT command using the following syntax

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

where:

ON

Enables the COMMIT and ROLLBACK commands for use with FOCUS data sources, and enables the use of the FOCUS Database Server to ensure transaction integrity.

OFF

Disables the COMMIT and ROLLBACK commands for use with FOCUS data sources, and disables the use of the FOCUS Database Server to ensure transaction integrity. This is the default.

Sharing Access to FOCUS Data Sources

The FOCUS DBMS ensures transaction integrity when multiple users are trying to access the same data source concurrently. If you are processing a transaction and—in the interval between beginning your transaction and completing it—the segments updated by your application have been changed and committed to the data source by another user, Maintain will roll back your transaction. This coordination is performed by the FOCUS Database Server. You can test to see if your transaction was rolled back by checking the value of the FocCurrent transaction variable, and then branch accordingly.

This strategy—in which FOCUS verifies that the records to which you wish to write have not been written to by another user in the interim—is called change verification. It allows many users to share write access to a data source, and grants update privileges for a given record to the first user that attempts the update.

Change verification takes advantage of the fact that two users rarely try to update the same record at the same time. Some DBMSs use strategies that lock out all but one user. Others grant update privileges to the first user that retrieves a record, even if he or she is the last one ready to update it—resulting in a performance bottleneck. In contrast, the FOCUS DBMS strategy of change verification enables the maximum number of users to access the same data concurrently, and makes it possible to write the maximum number of transactions in the shortest time. The FOCUS Database Server and the change verification strategy are designed for high-performance transaction processing.

How the FOCUS Database Server and Change Verification Work

The FOCUS Database Server's change-verification strategy is an extension of basic transaction processing. Each application user that accesses the FOCUS Database Server is known as a client. To ensure transaction integrity, follow this simple change-verify protocol:

1. As always, use the NEXT or MATCH commands to retrieve the data source records you need for the current transaction. When the application issues these commands, the server sends the application a private "client" copy of the records.
2. When the application issues a data source write command (such as INCLUDE, UPDATE, REVISE, or DELETE) against the retrieved records, it updates its private copy of the records.
3. When the application issues a COMMIT command to indicate the end of the transaction, the application's session sends a log of the transaction back to the server. The server now checks to see if any of the segments that the transaction changed have, in the interim, been changed and committed to the data source by other clients, and if any segments that the transaction added have, in the interim, been added by other clients. (You can customize which segments the FOCUS Database Server checks for changes by setting the PATHCHECK environment variable, as described in *Selecting Which Segments Will Be Verified for Changes* on page 2-47.)

The server takes one of the following actions:

- **No conflict.** If none of the records have been changed or added in the interim, then the transaction is consistent with the current state of the data source. The server writes the transaction to the data source and sets the application's FocCurrent transaction variable to zero to confirm the update.
 - **Conflict.** If any records have been changed in the interim, then the transaction might be inconsistent with the current state of the data source. The server ignores the transaction's changes to the data source—rolling back the transaction—and alerts the application by setting FocCurrent to a non-zero number.
4. The application evaluates FocCurrent and branches to the appropriate function.

Selecting Which Segments Will Be Verified for Changes

When you use a FOCUS Database Server, you can customize the change verification process by defining the segments for which the FOCUS Database Server will verify changes. You define this using the PATHCHECK environment variable.

You can choose between:

- **All segments in the path.** The FOCUS Database Server verifies that *all* segments in the path extending from the root segment to the target segment have not been changed and committed in the interim by other users. This is the default setting.
- **Modified segments only.** The FOCUS Database Server determines which segments you are updating or deleting, and verifies that those segments have not been changed and committed in the interim by other users.

You can set PATHCHECK for each FOCUS Database Server, which affects all applications that access FOCUS data sources managed by that FOCUS Database Server. To set it, issue the SET PATHCHECK command in the FOCUS Database Server profile (HLIPROF).

Syntax **How to Set PATHCHECK**

The PATHCHECK environment variable defines which segments the FOCUS Database Server will check for changes. To set PATHCHECK, issue the SET PATHCHECK command in the FOCUS Database Server profile (HLIPROF), using the following syntax

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

where:

ON

Instructs the FOCUS Database Server to verify that *all* segments in the path extending from the root segment to the target segment have not been changed and committed in the interim by other users. This is the default.

OFF

Instructs the FOCUS Database Server to check only segments that the current transaction has updated or deleted, and verify that those segments have not been changed and committed in the interim by other users.

Identifying the FOCUS Database Server

To identify which FOCUS Database Server will manage access to a given FOCUS data source, you must issue a USE command that associates the server with the data source. You can issue the USE command in a FOCUS profile procedure (FOCPROF or PROFILE), or at the beginning of the Maintain application's root procedure preceding the MAINTAIN command. For more information about FOCUS profile procedures, see the FOCUS installation guide for your operating environment.

Syntax

How to Identify a FOCUS Database Server With USE

For each FOCUS database that will be managed by a FOCUS Database Server, you must associate the database with the server in a USE command:

```
USE
datafile ON server_id
[datafile ON server_id]
.
.
.
END
```

where:

datafile

Is the file specification of a database to be managed by the FOCUS Database Server.

server_id

Is the ddname of the communication data set that points to the FOCUS Database Server job.

If you wish, you can identify multiple database/server pairs in one USE command.

Report Procedures and the FOCUS Database Server

When a FOCUS Database Server manages access to a FOCUS data source, each logical transaction that accesses that data source works with its own private copy of the data source's records. For more information about how a FOCUS Database Server manages access to a data source, see the *Simultaneous Usage Reference Manual* for your operating environment. This ensures that the transaction sees a consistent image of the data source that is isolated from changes being attempted by other users.

Non-Maintain procedures—for example, report procedures—are not part of a logical transaction; when control passes from a Maintain procedure to a non-Maintain procedure, the open transaction is suspended for the duration of the non-Maintain procedure. Therefore, if the non-Maintain procedure reports against a FOCUS data source, it accesses the live data source, not the open transaction's private copy. Changes made by the open transaction are not seen by the report, and changes committed by other users since the open transaction began are seen by the report, though not necessarily by the open transaction.

Sharing Data Sources With Legacy MODIFY Applications

A FOCUS data source being managed by a FOCUS Database Server can be accessed by both Maintain applications and legacy MODIFY applications. Note that while MODIFY allows creating records with duplicate keys, Maintain does not support FOCUS data sources that have duplicate keys.

Ensuring Transaction Integrity for DB2 Data Sources

DB2 ensures transaction integrity by locking data source rows when they are read. The behavior of a lock depends on a transaction's isolation level; the techniques discussed here for Maintain applications all use an isolation level of repeatable read. Repeatable read involves a trade-off: it ensures absolute transaction integrity, but it can prevent other users from accessing a row for long periods of time, creating performance bottlenecks.

Under repeatable read, a row is locked when it is retrieved from the data source, and is released when the transaction that retrieved the row is either committed to the data source or rolled back. A Maintain DB2 transaction is committed or rolled back each time a Maintain application issues a COMMIT or ROLLBACK command. You explicitly code COMMIT and ROLLBACK commands in your Maintain application; in some circumstances the application may also issue these commands implicitly, as described in *Designing Transactions That Span Procedures* on page 2-39, and in *When an Application Ends With an Open Transaction* on page 2-41.

We recommend two strategies for writing transactions to DB2 data sources:

- **Transaction locking.** This locks each row for the duration of the transaction—from the time a row is retrieved until the transaction is committed. In effect, it relies on DB2 to ensure transaction integrity. This is simpler to code, but keeps rows locked for a longer period of time. This is the preferred strategy, unless the duration of the locks interferes excessively with your data source concurrency requirements.
- **Change verification.** This locks each row while it is being retrieved, then releases the lock, and then relocks the row shortly before writing it to the data source. This technique ensures transaction integrity by verifying, before writing each row, that the row has not been changed by other users in the interim. This is more complex to code, but locks rows for a shorter period of time, increasing data availability.

While these strategies are described for use with DB2 data sources, you can also apply them to transactions against other kinds of data sources, changing DBMS-specific details when necessary.

Reference How Maintain's DB2 Logic Differs From Other IBI Products

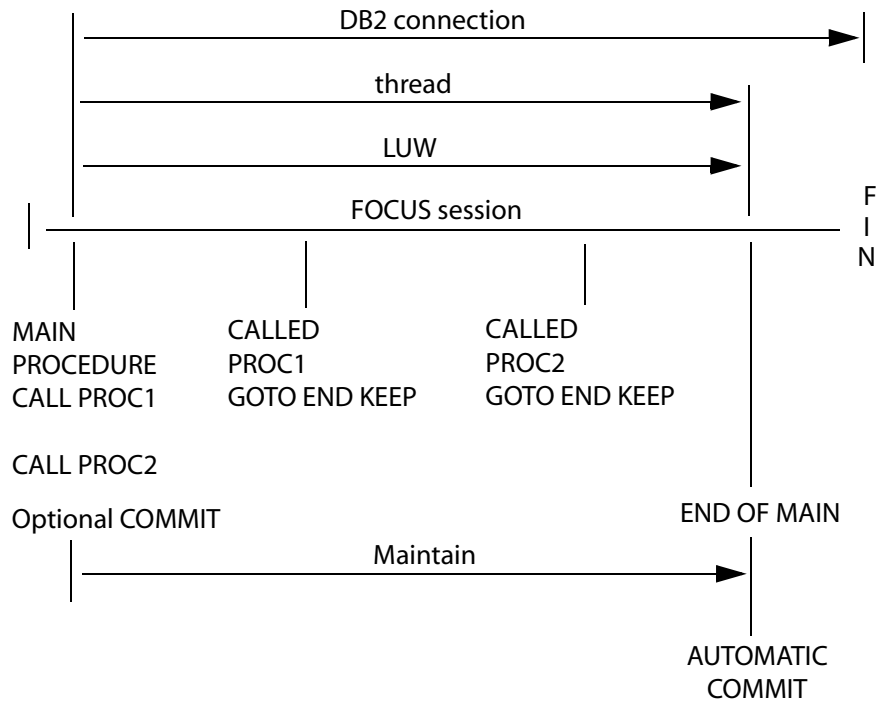
If you are familiar with using the DB2 Data Adapter with Information Builders products other than Maintain, note that Maintain works with DB2 a bit differently:

- Maintain enables you to issue COMMIT and ROLLBACK commands explicitly. It also issues them implicitly in certain situations, as described in *Designing Transactions That Span Procedures* on page 2-39, and in *When an Application Ends With an Open Transaction* on page 2-41.
- Maintain does not support the command SQL DB2 SET AUTOCOMMIT to control automatic commits.
- Because Maintain works on sets of rows, the FOCUS Interface to DB2 does not automatically generate change verification logic.

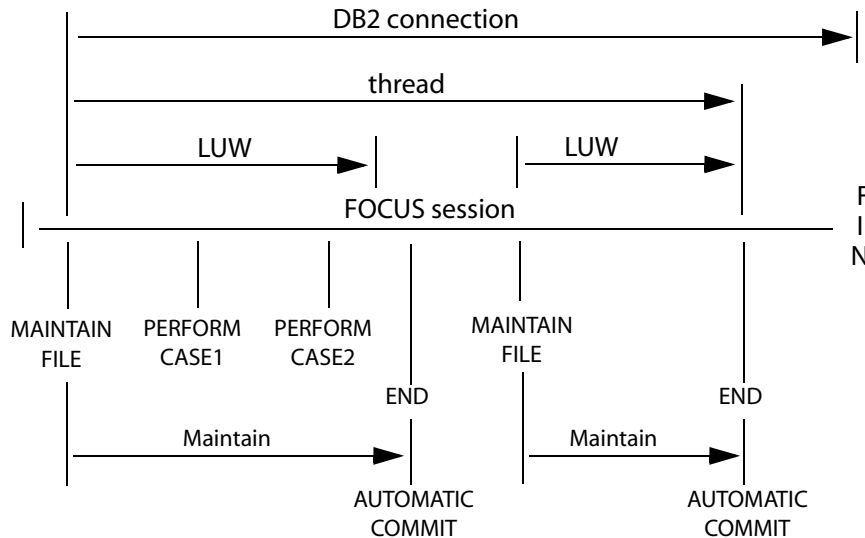
Using Transaction Locking to Manage DB2 Row Locks

You can use the transaction locking strategy to manage DB2 row locks in Maintain applications. When using transaction locking, your application locks each row with an isolation level of repeatable read for the duration of the transaction—from the time it retrieves the row until the time it commits or rolls back the transaction.

The following illustration shows the duration of connections, threads, and logical units of work when you use this strategy:



If your applications are small in scope, comprising only a single procedure, the duration of connections, threads, and logical units of work would look like this:



Compared to the change verification strategy, transaction locking is simpler to code, but keeps rows locked for a longer period of time. This may cause other users to experience time outs, in which case DB2 will return a -911 or -904 SQL code. You can mitigate the effect of row locking by:

- Keeping the size of the transaction small, making it less likely that another user will encounter a row locked by your transaction.
- Implementing the change verification strategy described in *Using Change Verification to Manage DB2 Row Locks* on page 2-55.
- Having user applications check for a locked condition when retrieving rows, and upon encountering a lock, re-issuing the retrieval request a specified number of times in a loop. If the user application exceeds the specified number of attempts, have it display a message to the user indicating that the row is in use, and suggesting that the user try again later.
- Using standard database administration techniques such as report scheduling, tablespace management, and data warehousing.

Procedure **How to Implement Transaction Locking for DB2**

To implement the transaction locking strategy for managing DB2 row locks in Maintain applications, bind the DB2 Interface plan with an isolation level of repeatable read. (The isolation level is a FOCUS Interface to DB2 installation BIND PLAN parameter.) In your Maintain application:

- 1. Read the rows.** Retrieve all required rows. Retrieval locks the rows with an isolation level of repeatable read.
- 2. Write the transaction to the data source.** Apply the transaction's updates to the data source.
- 3. Be sure to terminate called procedures correctly.** If a Maintain procedure calls another Maintain procedure within the scope of a transaction, the called procedure must return control using the GOTO END KEEP command. For more information about GOTO END KEEP, see *Designing Transactions That Span Procedures* on page 2-39.

Caution: If any called procedure within the scope of a transaction returns control without GOTO END KEEP, Maintain issues an implied COMMIT command, releasing all row locks and making the application vulnerable to updates by other users. Be sure to return control using GOTO END KEEP; otherwise, code each transaction within a single procedure, so that the scope of each transaction does not extend beyond one procedure, or use the change verification strategy described in *Using Change Verification to Manage DB2 Row Locks* on page 2-55.

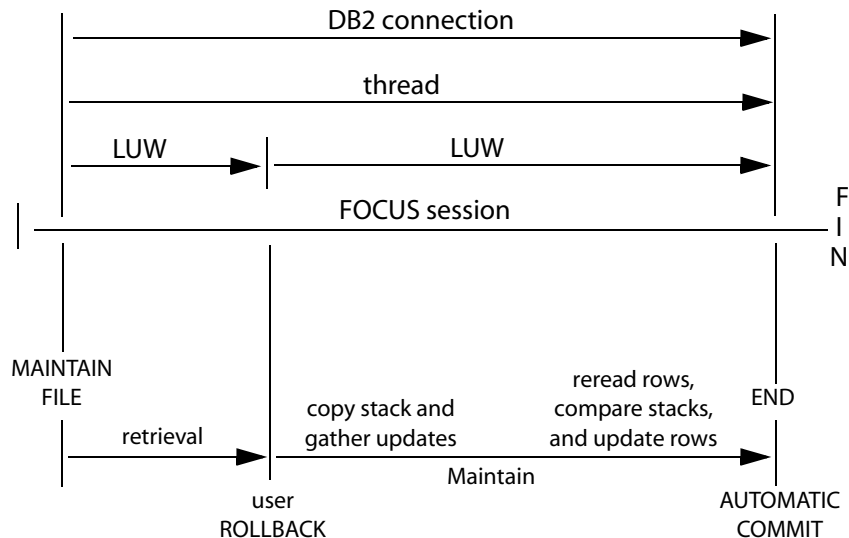
- 4. Close the transaction.** When the transaction is complete, close it by issuing a COMMIT or ROLLBACK command. The COMMIT or ROLLBACK command releases all row locks.

Using Change Verification to Manage DB2 Row Locks

You can use the change verification strategy to manage DB2 row locks in Maintain applications. When using change verification, your application retrieves all needed rows into a stack, locking them in the process; releases the locks after retrieval; and then performs all updates against the stack (*not* against the data source). This enables you to work with the data in the stack as long as necessary without preventing other users from accessing the data source. When you are ready to close the transaction, you re-retrieve the original rows from the data source, relocking them in the process. You then compare their current values in the data source to the values they held when you first retrieved them, and write the transaction to the data source if the values are the same—that is, if the rows have not been changed by other users in the interim.

Change verification enables the maximum number of users to access the same data concurrently, and makes it possible to write the maximum number of transactions in the shortest time. It is able to do this because it is an optimistic locking protocol—that is, it is optimized for the most common situation, in which at any moment, at most one user will attempt to update a given row. Compared to the transaction locking strategy, this is more complex to code, but locks rows for less time, increasing data availability.

The following illustration shows the duration of connections, threads, and logical units of work when you use this strategy:



Procedure How to Implement Change Verification for DB2

To implement the change verification strategy for managing DB2 row locks in Maintain applications, bind the DB2 Interface plan with an isolation level of repeatable read. (The isolation level is a FOCUS Interface to DB2 installation BIND PLAN parameter.) In your Maintain application:

- 1. Read the rows.** Retrieve all required rows into a stack (for example, Stack1). Retrieval locks the rows with an isolation level of repeatable read.
- 2. Free the row locks.** Issue a ROLLBACK command immediately following retrieval in order to release all row locks.
- 3. Copy the stack.** Make a copy of the stack (for example, Stack2). You will use this copy later when checking for changes.
- 4. Write the transaction to the stack.** Apply the transaction's updates to the rows in the original stack (Stack1).
- 5. Re-read the rows.** Re-retrieve the transaction's rows from the data source into a new stack (for example, Stack3). Retrieval relocks the rows with an isolation level of repeatable read.
- 6. Verify changes.** Compare the original data source values in the copy of the original stack (that is, Stack2) to the current data source values (that is, Stack3) to verify that other users have not changed these rows in the interim.
- 7. Write the transaction to the data source.** If any of these rows have been changed in the data source by another user, you can roll back the transaction or take some other action, as your application logic requires. If none of the rows in the transaction has been changed by other users in the interim, your application can apply the transaction's updates to the data source, and issue a COMMIT command to commit the transaction.

The COMMIT or ROLLBACK command releases all row locks.

Classes and Objects

Most application development is modular: the developer creates complex systems comprised of smaller parts. In conventional “procedural” development, these modules are processes (such as procedures), and data is defined within each process. In object-oriented development, the modules are models of real-world objects (such as a customer or a shipping order), and both data *and* processes are defined within each object. The object encapsulates the data and processes.

For example, if you are developing an order fulfillment system for a mail-order clothing business, the objects might include customers, orders, and stock items. A customer object’s data might include the customer’s ID code, phone number, and order history; the customer’s processes might include a function that adds the customer to a new mailing list, a function that updates the customer’s contact information, and a function that places an order for the customer.

Object-oriented development—because it models the real-world objects with which your enterprise deals, and encourages you to reuse application logic in a variety of ways—is a more efficient way to develop applications. Maintain enables you to create applications using object-oriented development, conventional development, or a hybrid of these two methods, providing you with a flexible development path.

What Are Classes and Objects?

Most applications need many objects of the same type. For example, if your business has 500 customers, you need one object to represent each customer. No one would want to design a customer object 500 times; clearly, you need a template that defines all customer objects, so that you can design the template once, and use it often. For example, you would use the template each time you create a new customer object to represent a new customer.

An object’s template is called its *class*. Each object is an instance of a class. In other words, the class defines the type of object. In fact, when you create a class, the class becomes a new data type. Just as you can use a built-in data type, such as integer or alphanumeric, to define a simple variable, you can use a class data type to define an object.

Example Comparing Classes and Built-in Data Types

Maintain supports two kinds of data types: built-in data types, and classes that you define yourself. For example, just as you can use the built-in data type alphanumeric to define a customer ID code

```
DECLARE CustID/A8;
```

you can use the class RetailCustomer to define an object as a customer:

```
DECLARE CustSmit8942/RetailCustomer;
```

Class Properties: Member Variables and Member Functions

You define a class by describing its properties. Classes have two kinds of properties:

- **Data**, in the form of the class's variables. Because these variables exist only as members of the class, they are called *member variables*. (In some object-oriented development environments these are also known as object attributes or instance variables.)

A class's member variables determine what the class *is* (as opposed to what it *does*). Each object of that class can have different values for its member variables.

- **Processes**, implemented as functions (which are also known as cases). Because these functions exist only as members of the class, they are called *member functions*. (In some object-oriented development environments these are also known as methods.)

A class's member functions define its behavior—that is, they determine what you can do to objects of that class, and in what ways you can manipulate their data.

Example Member Variables for a Customer Class

An application for a mail-order clothing business has defined a customer class named Customer. The class's member variables might include the customer's code, phone number, and most recent order number:

```
DESCRIBE Customer =  
(IDcode/A6,  
  Phone/I10,  
  LastOrder/A15);  
.  
.  
.  
ENDDESCRIBE
```

After declaring a new customer object for the customer Frances Smith

```
DECLARE CustFrSmith/Customer;
```

you could assign a value to Frances Smith's IDcode member variable:

```
DECLARE CustFrSmith.IDcode = GetNewCustCode();
```

Each object can have different values for its member variables; for example, in this case, each customer will have a different ID code.

Example Member Functions for a Customer Class

An application for a mail-order clothing business has defined a customer class named Customer. The class's member functions might include a function that adds the customer to a new mailing list, a function that updates the customer's contact information, and a function that places an order for the customer:

```
DESCRIBE Customer =
(IDcode/A6,
 Phone/I10,
 .
 .
 .
 LastOrder/A15);
CASE AddToList TAKES Name/A25, Address/A50, IDcode/A6;
.
.
.
ENDCASE
CASE UpdateContact ...
CASE PlaceOrder ...
ENDDESCRIBE
```

After declaring a new customer object for the customer Frances Smith

```
DECLARE CustFrSmith/Customer;
```

you could add Frances Smith to the mailing list using the AddToList member function:

```
CustFrSmith.AddToList();
```

Each object has the same member functions, and so the same behavior; for example, in this case, each customer will be added to the mailing list using the function.

Inheritance: Superclasses and Subclasses

If you want to create a new class that is a special case of an existing class, you could derive it from the existing class. For example, in a human resources application, a class called Manager could be considered a special case of a more general class called Employee: all managers are employees, and possess all employee attributes, plus some additional attributes unique to managers. The Manager class is derived from the Employee class, so Manager is a subclass of Employee, and Employee is the superclass of Manager.

A subclass inherits all of its superclass's properties (that is, it inherits all of the superclass's member variables and member functions). When you define a subclass, you can choose to override some of the inherited member functions; this means that you can recode them to suit the ways in which the subclass differs from the superclass. You can also add new member functions and member variables that are unique to the subclass.

Defining Classes

Before you can declare an object (an instance of a class), your procedure must have a class definition for that type of object. If the class:

- **Is already defined in a class library**, simply import the library into your procedure. (Class libraries are described in *Reusing Classes in Class Libraries* on page 2-62.)
- **Is already defined in another procedure**, simply copy and paste the definition into a class library; you can then import the library into any procedure that needs it.
- **Is not yet defined anywhere**, you can define it using the DESCRIBE command. Once it is defined, you can use it in that procedure, or copy it into a class library to be imported into multiple procedures.

Syntax **How to Define a Class or Subclass**

You can define classes (including subclasses) using the DESCRIBE command. You must issue the DESCRIBE command outside of a function—for example, at the beginning of the procedure prior to all functions. (Functions are also known as cases.)

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

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 6, *Language Rules Reference*.

superclass

Is the name of the superclass from which you wish to derive this class. Include only if this definition is to define 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 6, *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; for more information, see CASE in Chapter 7, *Command Reference*.

;

Terminates the definition if the definition omits member functions. If it includes member functions, the semicolon is omitted and the ENDDESCRIBE command 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.

Reusing Classes in Class Libraries

You can define a class once, but use it in multiple Maintain procedures, by storing its definition in a class library. A library is a kind of non-executable procedure in which you can store class definitions (as well as Maintain functions); you then import the library into each Maintain procedure in which you want to use those classes.

Procedure How to Create a Class Library

To create a class library:

1. Create a new Maintain procedure (that is, a FOCEXEC containing only MAINTAIN and END commands).
2. Create class definitions in the procedure. See *Defining Classes* on page 2-60 for more information about creating class definitions

or

Copy class definitions from other procedures and paste them into this procedure.

You can nest libraries to any depth using the MODULE IMPORT command. For example, to nest library B within library A, issue a MODULE IMPORT B command within library A. For more information about the MODULE IMPORT command, see *MODULE* in Chapter 7, *Command Reference*.

Note that a library cannot contain an explicit Top function.

Syntax How to Import a Class Library

You can use the MODULE command to import libraries containing class definitions so that the current procedure can use those classes. (Libraries can also contain other source code, such as function definitions.) The syntax is

```
MODULE IMPORT (library_name [, library_name] ... );
```

where:

library_name

Is the name of the Maintain procedure that you wish to import as a source code library. Specify its file name without an extension.

The library is a FOCEXEC file, and its naming and allocation requirements are those for FOCEXEC files generally.

The MODULE command must immediately follow the procedure's MAINTAIN command.

Declaring Objects

Once a class definition exists, you can declare objects of that class. This is identical to declaring the simple variables of a built-in data type. You declare objects using the DECLARE command.

Syntax How to Declare an Object

You can declare a local or global object using the DECLARE command. To make the declaration:

- **Local**, code the DECLARE command in the function to which you want it to be local, following the function's CASE command, and preceding all the other commands in the function.
- **Global**, code the DECLARE command outside of any function (for example, at the top of the procedure, following the MAINTAIN command and any MODULE IMPORT commands, where it will be easy to find).

You can also create global objects using the COMPUTE command. For information about the COMPUTE command, see *COMPUTE* in Chapter 7, *Command Reference*.

To declare an object using the DECLARE command, use this syntax

```
DECLARE
[ (
objectname/class [= expression];
.
.
.
) ]
```

where:

objectname

Is the name of the object that you are creating. The name is subject to the Maintain language's standard naming rules; for more information, see *Specifying Names* in Chapter 6, *Language Rules Reference*.

class

Is the name of the class of which this object will be an instance.

expression

Is an optional expression that will provide the object's initial value. If the expression is omitted, the object's initial value is the default for that data type: a space or null for date and alphanumeric data types, and zero or null for numeric data types.

()

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

CHAPTER 3

Tutorial: Coding a Procedure

Topics:

- Step 1: Beginning and Ending the Procedure
- Step 2: Selecting Records
- Step 3: Collecting Transaction Values
- Step 4: Writing Transactions to the Data Source
- Step 5: Issuing the Procedure
- Step 6: Browsing Through a Stack and Using Triggers
- Step 7: Displaying and Editing an Entire Stack in a Winform

Maintain is a rich data maintenance language that offers many sophisticated features. However, the basic design is simple enough so that you can quickly learn the basic concepts and syntax and begin developing useful applications right away.

The following topics introduce you to Maintain's basic concepts and syntax. As you follow the tutorial's step-by-step approach, you build a simple Maintain procedure and become familiar with:

- How to perform fundamental data maintenance operations.
- How different operations function together within a Maintain procedure.
- What the equivalent operations are in a MODIFY request.

These topics describe the following data maintenance operations:

- Selecting records.
- Collecting transaction values.
- Manipulating stacks and fields.
- Controlling the flow of a procedure.
- Writing transactions.
- Issuing the procedure.

Each step is divided into three parts:

- A **Goal** that briefly states what you want to accomplish in this part of the procedure.
- **Methods** that you can use to achieve your goal. This part provides you with task-oriented instructions for using Maintain. If you are in a hurry to build the application, you can skim or skip this section and move on to the Solution section.
- A **Solution** that implements a method and shows the resulting code used to build the sample application. At each step in building the procedure, the code developed so far appears in a box, and the commands added during that step appear in **bold**.

The following topics are not intended to provide the complete syntax or explanations of Maintain commands, functions, and variables. For a complete reference see Chapter 6, *Language Rules Reference*; Chapter 7, *Command Reference*; Chapter 8, *Expressions Reference*, and *Using Functions*.

In these topics and in Chapter 4, *Tutorial: Painting a Procedure*, the instruction **Try it now:** indicates an instruction for you to type text, select from a dialog box, or press a key.

Two Ways to Follow the Tutorial

This tutorial builds a sample application in three stages. Each stage of the application is stored in a pair of FOCEXEC and WINFORMS files (named VIDTAPE1, VIDTAPE2, and VIDTAPE3) that are supplied with FOCUS.

There are two ways to use this quick-start tutorial:

- Follow the tutorial's steps, entering the code for each step into a FOCEXEC using an editor such as TED. When the instructions tell you to run the application, run the version that you coded.

To follow the tutorial this way, you must create your own set of application files:

- 1.** Choose a name for the application (for example, TUTOR).
 - 2.** Create a new FOCEXEC file using your personal application name and append the number 1 to the name (for example, TUTOR1 FOCEXEC). You enter code into this file during the tutorial.
 - 3.** Copy the three VIDTAPE# WINFORMS files that are supplied with FOCUS and rename the copies with your new application name (for example, TUTOR1, TUTOR2, and TUTOR3 WINFORMS).
- Follow the steps of the tutorial without entering any code. When the instructions tell you to run the application, run the version that is supplied with FOCUS.

Building the Sample Application

Beginning with Step 1, you build a sample application containing a Maintain procedure that accesses the VideoTrk data source. This data source contains customer, sales, and rental information for a video rental shop. You build the procedure in stages, and each stage illustrates a few basic concepts.

The initial goal is to create a procedure that:

1. Identifies customers whose membership is expiring during the first three weeks of June.
2. Enables a shop clerk to edit membership and mailing information for these customers using a form displayed on a screen.
3. Updates the data source with the edited customer information.

After you build this basic application, you enhance it so that:

4. The clerk can browse through the customer list and display and edit information for any customer.

Finally, you expand the application so that:

5. For each customer, the clerk can display and edit information about all of the video tapes that the customer rented.

This sample illustrates basic Maintain processing. It does not illustrate event-driven application development, which is presented in Chapter 4, *Tutorial: Painting a Procedure*. Chapter 4, *Tutorial: Painting a Procedure* is intended as the second phase of your Maintain training and assumes that you are already familiar with the material presented in Chapter 2, *Maintain Concepts*, and in this tutorial.

This application writes to the VideoTrk sample data source. If you need to create this data source, see Appendix A, *Master Files and Diagrams*. If you do not have write-access to VideoTrk, consult with the database administrator.

Step 1: Beginning and Ending the Procedure

Goal

Before you develop the logic of the procedure, you must know how to begin and end it.

Methods: MAINTAIN and END Commands

All Maintain procedures begin and end with two simple commands (which must be in uppercase) as illustrated in the following syntax definition:

```
MAINTAIN FILE filename
.
.
.
END
```

If you wish to work with several data sources in one procedure, you specify all of them in the MAINTAIN command. Unlike the MODIFY facility, the COMBINE command is not required. For example:

```
MAINTAIN FILES Employee AND EducFile AND JobFile
.
.
.
END
```

The MAINTAIN command has additional phrases used in other types of situations. As with all Maintain commands, the complete syntax is described in Chapter 7, *Command Reference*.

Solution

You now have the beginning and end of the sample Maintain procedure:

```
MAINTAIN FILE VideoTrk
END
```

Try it now: If you have created your own set of application files, enter those commands into your Maintain procedure file, for example, into TUTOR1 FOCEXEC.

Step 2: Selecting Records

Goal

To update customer records, you first must identify and retrieve the records. You use the NEXT command to select the records and copy the relevant information (customer ID, name, address, and phone number) into the CustInfo stack.

Methods: NEXT and MATCH

To read data from a data source, you first must select the record where the data resides. In MODIFY, there were two ways of doing this:

- By field value—one segment instance at a time, one record at a time—using MATCH.
- Sequentially—one segment instance at a time, one record at a time—using NEXT.

For example, before MODIFY displays a record in a CRTFORM or writes it to the SPA, you must select it using MATCH or NEXT. You also must identify the record before MODIFY includes, updates, or deletes it. Thirdly, you can only identify one segment instance, in one record, at a time.

Maintain releases you from these restrictions. In Maintain, there are five ways of selecting records:

- **By field value—for an entire set of records**—using NEXT. The WHERE phrase enables you to select by value, and the FOR ALL phrase selects the entire set of records that satisfies the WHERE selection condition. The basic syntax for this is:

```
FOR ALL NEXT fields INTO stack WHERE selection_condition;
```

- **By field value—for a sequence (subset) of records**—using NEXT. This method is similar to selecting by field value for an entire set except that it employs the FOR *n* phrase, selecting—at the current position in the data source—the first *n* records that satisfy the WHERE condition. The basic syntax for this is:

```
FOR n NEXT fields INTO stack WHERE selection_condition;
```

- **By field value—one segment at a time, one record at a time**—using MATCH. This method works very much like MATCH in MODIFY, except that the values on which you are matching can be taken from a source stack, and the fields that you retrieve can be placed into a destination stack. The basic syntax for this is:

```
MATCH fields [FROM stack] [INTO stack];
```

- **Sequentially—for a sequence (subset) of records**—using NEXT. This method employs the FOR *n* phrase to select the next *n* records. The basic syntax for this is:

```
FOR n NEXT fields INTO stack;
```

- **Sequentially—one segment instance at a time, one record at a time**—using NEXT. This method works very much like NEXT in MODIFY, except that the fields which you retrieve can be placed into a destination stack. The basic syntax for this is:

```
NEXT fields [INTO stack];
```

Specifying Data Source Position With the REPOSITION Command

Each time you issue a NEXT command, Maintain begins searching for records from the current position in the data source. For example, if your first data source operation retrieved a set of records:

```
FOR ALL NEXT CustID INTO SmokeStack
WHERE ProdName EQ 'VCR DUST COVER';
```

Maintain has searched sequentially through the entire data source. The current position marker now points to the end of the data source. If you issue another NEXT command:

```
FOR ALL NEXT LastName FirstName INTO CandyStack
WHERE ProdName EQ 'CANDY';
```

Maintain searches from the current position to the end of the data source. Because the current position is the end of the data source, no records are found.

When you want a NEXT command to search through the entire data source (often the case when you wish to retrieve a set of records), you must first issue the REPOSITION command to move the current position marker to the beginning of the data source.

For example, the following REPOSITION command specifies the CustID field—which is in the root segment—and so moves the current position marker for the root segment chain and all of its descendant chains back to the beginning of the chain (in effect, back to the beginning of the data source):

```
REPOSITION CustID;
FOR ALL NEXT LastName FirstName INTO CandyStack
WHERE ProdName EQ 'CANDY';
```

REPOSITION is similar to the MODIFY command of the same name.

Solution

You add a NEXT command to retrieve the desired records. Each part of the command plays a different role:

- **FOR ALL** selects a set of records.
- **WHERE ExpDate GE 920601 AND ExpDate LE 920621** restricts the set to the customers whose memberships are expiring during the first three weeks of June.
- **INTO CustInfo** copies all the fields from the specified segments—in this case, the root segment—into the stack named CustInfo.

The procedure now looks like this:

```
MAINTAIN FILE VideoTrk  
  
FOR ALL NEXT CustID INTO CustInfo  
WHERE ExpDate GE 920601 AND ExpDate LE 920621;  
  
END
```

Try it now: If you have created your own set of application files, enter these additional commands into your FOCEXEC file.

Step 3: Collecting Transaction Values

Goal

You now have a set of customer records—the CustInfo stack—to display so that the clerk can update them. You do this using a Winform.

Methods: WINFORM and NEXT

After you select data source records, you can apply transaction values to update or delete these records. You also can add new records without performing any prior record selection.

There are two ways of collecting transaction values:

- **Interactively at the screen.** Users can enter values into Winforms using the WINFORM command which provides a sophisticated graphical user interface.
- **From a transaction file.** Your procedure can read a file of transaction values using NEXT. These files provide a handy way of collecting transaction values from another application.

NEXT is an expanded version of the MODIFY command of the same name. Its primary role (selecting records) was described in Step 2. NEXT also replaces MODIFY's FIXFORM command.

WINFORM replaces the MODIFY CRTFORM command and provides a user interface that is more powerful and easier to use.

Solution

You display a Winform named ShowCust that displays the CustInfo stack and enables the clerk to edit membership, address, and phone information in the stack. For information about designing and creating Winforms and a tutorial, see Chapter 4, *Tutorial: Painting a Procedure*.

The Painter also adds a comment line which states

```
>> Generated Code Section
```

Do not add or edit any lines following the comment line. The Painter uses this section (from this line through the end of the procedure) for Maintain functions that it generates. (Maintain functions are often referred to more simply as functions, and are also known as cases.) It also uses this section for comments describing the properties and layout of the Winform, the Winform's triggers, and stack and field bindings. These comments document your application and are explained in Chapter 5, *Using the Winform Painter*.

Note that if you have created your own set of application files, you do not see the additional comments after the comment line just described.

You can omit the comments from your Maintain procedure by selecting the *Preferences* option from the File menu in Winform Painter and then deselecting *Pictorial View*. In the interest of simplicity, as the comments do not affect the logic of the procedure, they are not shown in the following sample code.

Note that the Winform Painter changes the VideoTrk specification in the MAINTAIN FILE command to all uppercase.

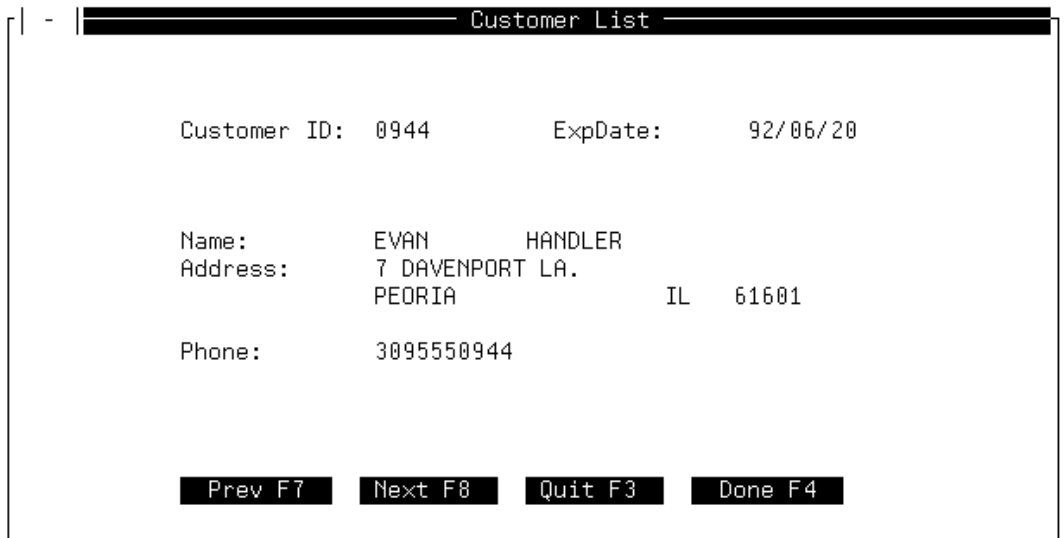
The procedure now looks like this:

```
MAINTAIN FILE VIDEOTRK
FOR ALL NEXT CustID INTO CustInfo
WHERE ExpDate GE 920601 AND ExpDate LE 920621;
WINFORM SHOW ShowCust;
-* >> Generated Code Section....
.
.
.
END
```

Try it now: If you have created your own set of application files, enter these additional commands into your Maintain procedure file.

Step 3: Collecting Transaction Values

The ShowCust Winform already has been created and is in the VIDTAPE1 WINFORMS file. You can view it after you run the application in *Step 5: Issuing the Procedure* on page 3-13.



The screenshot shows a window titled "Customer List" with the following data:

Customer ID:	0944	ExpDate:	92/06/20
Name:	EVAN HANDLER		
Address:	7 DAVENPORT LA. PEORIA IL 61601		
Phone:	3095550944		

At the bottom of the window, there are four buttons: "Prev F7", "Next F8", "Quit F3", and "Done F4".

Step 4: Writing Transactions to the Data Source

Goal

Now that the customer information has been edited, you must write the updated information to the data source. You do this using the UPDATE and COMMIT commands.

Methods: Write Commands, COMMIT, and ROLLBACK

To make changes to a data source, you issue the write commands: INCLUDE, UPDATE, REVISE, or DELETE. These perform the same functions as the MODIFY commands of the same names. In Maintain, however, these commands are enhanced so that you can perform a transaction for:

- A set of records, using the FOR ALL prefix and the FROM phrase.
- A sequence (subset) of records, using the FOR *n* prefix and the FROM phrase.

These phrases were described for other commands in *Step 2: Selecting Records* on page 3-6 and *Step 3: Collecting Transaction Values* on page 3-8.

To guarantee the integrity of all the include, update, and delete operations comprising a single logical transaction, you code a COMMIT command immediately following the last step of the transaction. Then, if a system problem or another application interrupts the logical transaction, it is rolled back—that is, cleared—without being written to the data source. In this way, you can guarantee that no transaction component is ever written to the data source unless all transaction components are—that is, unless it is part of a complete and successful logical transaction.

Solution

You add an UPDATE command to update the data source. Each part of the command plays a different role:

- **FOR ALL** selects all of the rows in the stack with which to update the data source.
- **FROM CustInfo** takes the rows from the stack named CustInfo. The columns specified in the command are used to update the data source.

You define the entire procedure as one logical transaction and issue a COMMIT command to send the changes to the data source. The first stage of the procedure is now finished and looks like this:

```
MAINTAIN FILE VIDEOTRK

FOR ALL NEXT CustID INTO CustInfo
WHERE ExpDate GE 920601 AND ExpDate LE 920621;

WINFORM SHOW ShowCust;

FOR ALL UPDATE LastName FirstName Street City State Zip
Phone FROM CustInfo;

COMMIT;

-* >> Generated Code Section...
.
.
.
END
```

Try it now: If you have created your own set of application files, enter these additional commands into your Maintain procedure file.

Step 5: Issuing the Procedure

Goal

Now that you have completed the first stage of the procedure, you can issue it.

Methods: **CALL, COMPILE, RUN**

There are three ways of issuing a Maintain procedure:

- **From the TED text editor.** You can enter the procedure as a FOCEXEC in TED and then issue the RUN or MNTCON RUN command at the TED command line.
- **At the FOCUS command line.** You can save the procedure as a FOCEXEC and enter the EX or MNTCON EX command to execute it at the FOCUS command line.
- **Within another Maintain procedure.** You can save the procedure as a FOCEXEC and from a second procedure, enter the CALL command to execute it.

You can run a procedure more efficiently, increasing the speed of execution, by first compiling it using the COMPILE or MNTCON COMPILE commands. The commands compile both the Maintain procedure and—if one exists—the associated Winform, generating a FOCCOMP file. You can run a compiled Maintain procedure by issuing the RUN or MNTCON RUN command.

Solution

You use the MNTCON EX command to run the procedure. You can run the completed procedure which is stored as VIDTAPE1 FOCEXEC on the release tape or supply the name of the Maintain procedure where you stored your code.

Try it now: If you have created your own set of application files:

1. Enter *FILE* at the TED command line to save your file and exit TED.
2. Enter the following at the FOCUS command prompt (substituting the name of your Maintain procedure file for VIDTAPE1):

```
MNTCON EX VIDTAPE1
```

The ShowCust Winform appears.

The screenshot shows a terminal window titled "Customer List". The window displays the following information:

```
Page _ 1 of 16  
  
Customer ID: 0944      ExpDate: 92/06/20  
  
Name:      EVAN      HANDLER  
Address:   7 DAVENPORT LA.  
          PEORIA      IL  61601  
  
Phone:    3095550944  
  
[ Prev F7 ] [ Next F8 ] [ Quit F3 ] [ Done F4 ]
```

Step 6: Browsing Through a Stack and Using Triggers

Goal

In the first stage of this application, you used a Winform that displays the CustInfo stack. However, this Winform displays only one row of the stack and does not offer a way of moving through the stack to display different rows.

Now you enhance the application so the clerk can browse through the stack by using triggers. A trigger is logic that is invoked (“triggered”) by a specified event. Each time that the event occurs, the logic is invoked. In Maintain, the invoked logic is a function or a system action, and the event is something the user does in a Winform. (Triggers are also known as event handlers.)

Methods: Winform Painter, Triggers, IF, FocIndex, FocCount

You can use the Winform Painter to generate a stack browser in your Winform, as described in Chapter 5, *Using the Winform Painter*. The Winform facility manages the browser automatically without inserting any code into the Maintain procedure.

To illustrate the principles of stack manipulation for this sample application, you code your own browser. In addition to making the browser mechanism visible, this also enables you to enhance the basic browser in *Step 7: Displaying and Editing an Entire Stack in a Winform* on page 3-18.

Your browser uses the IF command, together with the FocIndex and FocCount stack variables, to loop through the stack. You put this logic into a function and then assign the function to a function-key trigger. Whenever a user presses a specified function key, the trigger invokes the function, and the logic in the function moves through the stack one row at a time.

The basic logic, shown here for the CustInfo stack, is:

```
IF CustInfo.FocIndex LT CustInfo.FocCount
THEN COMPUTE CustInfo.FocIndex = CustInfo.FocIndex + 1;
```

The IF command tests whether the current position in the stack (FocIndex) has reached the last record in the stack (FocCount). If it has not, then it can continue to move forward through the stack one row at a time, so it increases the current position by one. When control leaves the function and returns to the Winform, the next row is displayed.

If, when browsing through the stack, you wish to return to the first row after displaying the last row, you can add the following ELSE phrase to the generated IF command:

```
ELSE COMPUTE CustInfo.FocIndex = 1;
```

Solution

You implement the stack browser by coding the following two functions. The `NextCustomer` function moves forward through the stack one row at a time; the `PreviousCustomer` function moves backward through the stack one row at a time.

Note that the browser functions are not called by any code in the procedure but instead are invoked by a trigger from the Winform. (In this case, a user selects the *Prev* or *Next* buttons on the screen or presses the corresponding F7 or F8 keys.) After the function executes, control returns to the Winform. When the user exits the Winform (by pressing PF4), control passes to the command that follows the `WINFORM SHOW` command.

```
MAINTAIN FILE VIDEOTRK

FOR ALL NEXT CustID INTO CustInfo
    WHERE ExpDate GE 920601 AND ExpDate LE 920621;

WINFORM SHOW ShowCust;

FOR ALL UPDATE FirstName LastName Street City State Zip
    Phone FROM CustInfo;

COMMIT;

CASE NextCustomer
    IF CustInfo.FocIndex LT CustInfo.FocCount THEN
        COMPUTE CustInfo.FocIndex = CustInfo.FocIndex + 1;
    ENDCASE

CASE PreviousCustomer
    IF CustInfo.FocIndex GT 1 THEN
        COMPUTE CustInfo.FocIndex = CustInfo.FocIndex - 1;
    ENDCASE

-* >> Generated Code Section....
.
.
.
END
```

Try it now: If you have created your own set of application files:

1. Make a copy of your Maintain procedure file, rename it substituting "2" for "1" (for example, TUTOR2 FOCEXEC), and enter these additional commands into it.
2. Enter *FILE* at the TED command line to save your file and exit TED.
3. run the new procedure (either the version supplied with FOCUS, named VIDTAPE2 or your personal version that you named) at the FOCUS command line.

The Winform appears. Four function keys appear at the bottom of the screen:

- **F7** triggers the PreviousCustomer function.
- **F8** triggers the NextCustomer function.
- **F3** triggers the Exit system action.

It immediately exits both the Winform and the Maintain procedure.

If you trigger an immediate exit from the procedure, control does not continue to the UPDATE command, and the data source is not updated.

- **F4** triggers the Close Winform system action. It exits the Winform and returns to the Maintain procedure.

Notice that Exit and Close Winform do not have functions in the procedure. They are provided by the Winform facility; no code must be generated for them.

```

| - |----- Customer List -----|
| Page _ 1 of 15 |
|
| Customer ID: 0944      ExpDate: 92/06/20
|
| Name:      EVAN      HANDLER
| Address:   7 DAVENPORT LA.
|           PEORIA      IL  61601
|
| Phone:    3095550944
|
|-----|-----|-----|-----|
| Prev F7 | Next F8 | Quit F3 | Done F4 |
|-----|-----|-----|-----|

```

Step 7: Displaying and Editing an Entire Stack in a Winform

Goal

In the final stage of the application, you enable the clerk to display and edit information about each customer's video rentals. Adding this functionality illustrates how to use multiple stacks and how to work with stack editors—called grids—to display and edit all the rows of a stack at the same time within a Winform.

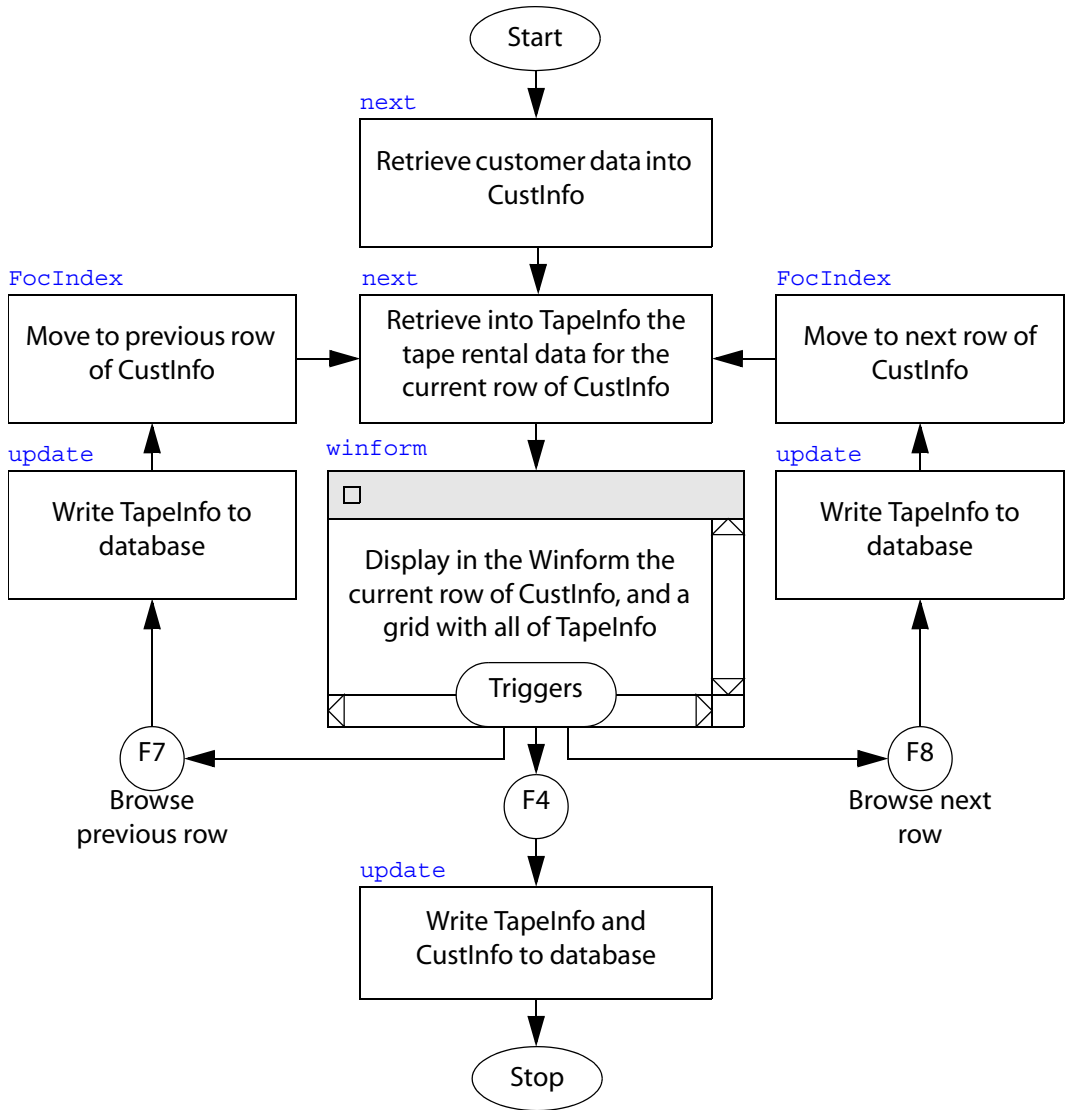
Methods: Multiple Stacks and Stack Editors (Grids)

Up until now you have worked with procedures that use only a single stack. You can develop more powerful applications by using multiple stacks. For example, to achieve your current goal, the application keeps your CustInfo stack—which contains each customer's membership information, name, address, and phone number—and creates a second stack, named TapeInfo, to hold one customer's videotape rental information.

For each row of CustInfo (representing one customer), the application retrieves all of the customer's tape rental data into TapeInfo. When the clerk finishes editing the data in TapeInfo, the application writes the new data to the data source, clears the stack, and then retrieves into it the tape rental data for the next customer.

However, to realize fully the power of stacks and set-based processing, you want to display and edit an entire stack at one time. You can do this by placing a stack editor—called a grid—in a Winform. The grid enables you to see multiple rows and columns at one time. If there are too many rows or columns to fit within the grid box, the Winform facility automatically provides scroll bars to scroll the hidden elements into view.

The following chart illustrates how you combine your existing application with multiple stacks and a grid to achieve your goal:



Solution

You add a function called `GetRental` to the version of the procedure in Step 6. For a given row of `CustInfo`, `Get Rental` retrieves tape rental data into `TapelInfo`. (In the `WHERE` phrase in `GetRental`, `CustInfo().CustID` refers to the value of `CustID` in the current row of `CustInfo`.) Each time the function is invoked, it clears the current values of `TapelInfo`, repositions `VideoTrk` to the beginning of the data source, and retrieves the records. Note that the default value of `FocIndex` is 1; the first time that `GetRental` is invoked, it retrieves rental data for the first customer in `CustInfo`.

You also add a command that performs a new function—`UpdateRental`—to the browser functions, so that when the clerk triggers them from the Winform, the application writes the updated rental information to the data source before advancing to the next customer in the `CustInfo` stack.

Because you are no longer interested only in customers whose membership is expiring but rather in all customers, you remove the `WHERE` phrase from the first `NEXT` command.

The final stage of the application follows:

```
MAINTAIN FILE VIDEOTRK

FOR ALL NEXT CustID INTO CustInfo;

PERFORM GetRental;

WINFORM SHOW ShowCust;

FOR ALL UPDATE FirstName LastName Street City State Zip
    Phone FROM CustInfo;

PERFORM UpdateRental;

CASE GetRental
    STACK CLEAR TapeInfo;
    REPOSITION CustID;
    FOR ALL NEXT CustID TransDate MovieCode INTO TapeInfo
        WHERE VideoTrk.CustID EQ CustInfo().CustID;
ENDCASE

CASE UpdateRental
    FOR ALL UPDATE ReturnDate Fee FROM TapeInfo;
    COMMIT;
ENDCASE

CASE NextCustomer
    PERFORM UpdateRental;
    IF CustInfo.FocIndex LT CustInfo.FocCount THEN
        COMPUTE CustInfo.FocIndex = CustInfo.FocIndex + 1;
    PERFORM GetRental;
ENDCASE
```

```

CASE PreviousCustomer
  PERFORM UpdateRental;
  IF CustInfo.FocIndex GT 1 THEN
    COMPUTE CustInfo.FocIndex = CustInfo.FocIndex - 1;
  PERFORM GetRental;
ENDCASE
-* >> Generated Code Section....
.
.
.
END

```

Try it now: If you created your own set of application files:

1. Make a copy of your second Maintain procedure file, rename it substituting "3" for "2" (for example, TUTOR3 FOEXEC), and enter these additional commands into it.
2. Enter *FILE* at the TED command line to save your file and exit TED.
3. run the new procedure (either the version supplied with FOCUS, named VIDTAPE3 or your personal version that you named) at the FOCUS command line.

The new Winform appears. The grid is located in the center of the form.

- |
Customer List

Page 1 of 36

Customer ID: 0925

Name: IVY CRUZ

Address: 86 ELLIOTT AVE.
PEORIA IL 61601

Phone: 3095550925

MOVIECODE	COPY	RETURNDATE	FEE
001MCA	1	91/06/29	2.00
692PAR	2	91/06/29	2.00

Prev F7
Next F8
Quit F3
Done F4

Step 7: Displaying and Editing an Entire Stack in a Winform

CHAPTER 4

Tutorial: Painting a Procedure

Topics:

- Step 1: Creating a New Winform
- Defining the Winform's Properties
- Saving Your Work and Exiting
- Step 2: Adding Fields
- Step 3: Adding a Grid
- Step 4: Adding Text
- Step 5: Adding Buttons and Triggers
- Step 6: Coding Triggers and Other Functions
- Step 7: Running the Maintain Request

In Chapter 3, *Tutorial: Coding a Procedure*, you developed a Maintain application. To simplify the tutorial and focus on basic transaction logic, you were provided with the completed Winforms. Now that you are familiar with coding a basic request, in this tutorial you will develop a complete application. You paint a Winform, generate code, and add triggers all in a single application development environment: the Winform Painter.

The following topics teach you how to use many of the Painter's features and give you hands-on experience with event-driven processing and event-driven development. As you follow the tutorial's step-by-step approach, you learn how to:

- Begin a Painter session and create a new procedure that includes a Winform.
- Add fields, grids, text, and buttons to the Winform.
- Create triggers to respond to a user's actions.
- Change previous design work.
- Generate and supply code.

In these topics and in Chapter 3, *Tutorial: Coding a Procedure*, "**Try it now:**" introduces an instruction for you to type text, select from a dialog box, or press a key.

Step 1: Creating a New Winform

First, you learn about the application you are developing and how to begin a Winform Painter session. Then, you create a new procedure that includes a Winform.

Description of the Application

The application you develop in this tutorial writes to the Car sample data source. If you do not have write access to Car, consult with the database administrator.

You build an application user interface, that is, a Winform, that looks like the following:

```
Car Editing Form
Country      : ENGLAND
Car          : JAGUAR

MODEL        BODYTYPE    DEALER_COST
1 V12XKE AUTO CONVERTIBLE  7,427
2 XJ12L AUTO  SEDAN       11,194

Select country and car, then update old cost data

Previous Car F1  Next Car F2  Quit F3  Done F4
```

The Country and Car fields appear at the top of the screen. Below these two fields, a grid displays multiple rows of the Model, BodyType, Dealer_Cost, and Retail_Cost stack columns.

You can press PF1 and PF2 to scroll through the Country and Car values. Inside the grid, you can use scroll bars or function keys to scroll the data backward (PF7) and forward (PF8).

How to Open the Painter

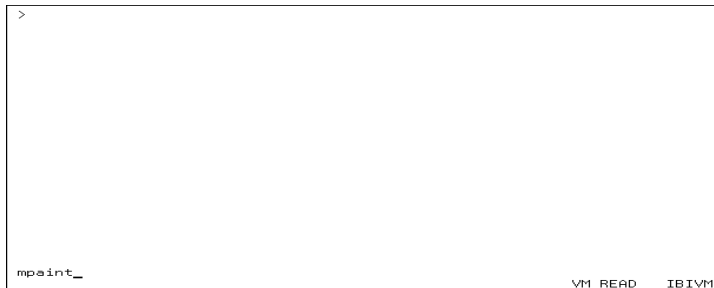
To open Maintain's Winform Painter, you enter MPAINT at the FOCUS prompt.

You can provide the FOCEXEC name that displays the Winforms you want to paint. The syntax is:

```
MPAINT procedure_name
```

For the purposes of this tutorial, do *not* supply the Maintain procedure name.

Try it now: Enter MPAINT at the FOCUS prompt.



The Open Maintain dialog box appears.

If your screen does not display scroll bars similar to the ones in the Open Maintain dialog box shown in *Naming the Procedure* on page 4-6, or if the check boxes in subsequent screens do not look like the one in the screen shown in *System Setup Dialog Box* on page 4-4, exit the Winform Painter. Follow the instructions in *Adjusting Winform Appearance* on page 4-3 to specify the desired appearance.

If your screens appear similar to the screen captures, you can skip to *The Painter Dialog Boxes* on page 4-6.

Adjusting Winform Appearance

Different terminals and terminal emulators process screen information in different ways. Some configurations support dashed lines instead of solid Winform borders. For solid borders, SBORDER must be set to ON in your FOCUS session. (See the *Developing Applications* manual for information about SET commands.)

You can change the Winform Painter's terminal emulator setting to display the options that appear on the screen.

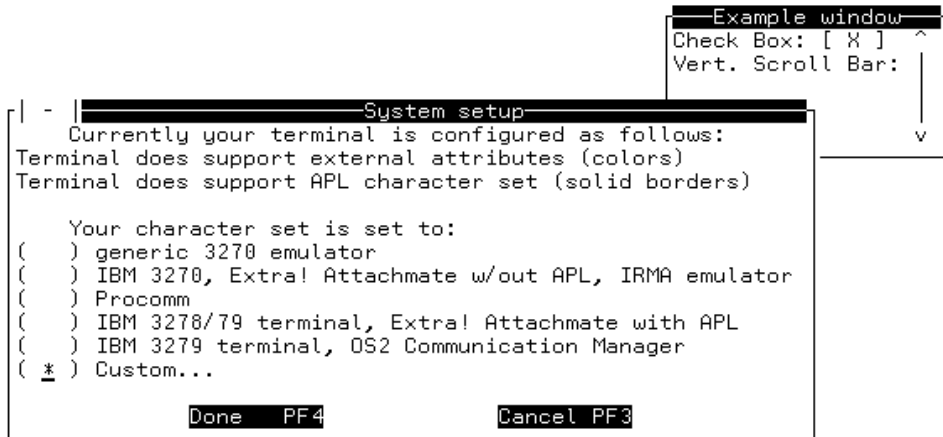
Reference System Setup Dialog Box

Try it now: To change the terminal emulator setting, at the FOCUS command line, enter:

EX MSETUP

or, from the File menu while in the Painter, select *Preferences* and then *Terminal*.

FOCUS displays the following System setup dialog box and an Example window:



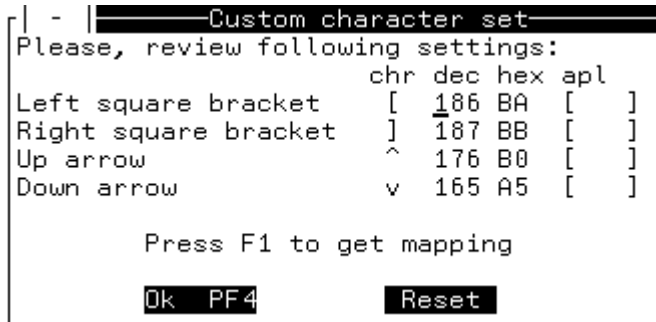
Five terminal emulations and an option for customized emulation are available. The Example window in the upper right-hand corner of the screen provides examples of the check box and the scroll bar for the currently selected emulation.

Try it now: To select your terminal emulation:

1. Position the cursor in the appropriate box and press any key.
2. Press Enter. (To cancel what you have entered, press PF3.)
3. When you are finished, press PF4.

The box you selected displays an asterisk (*).

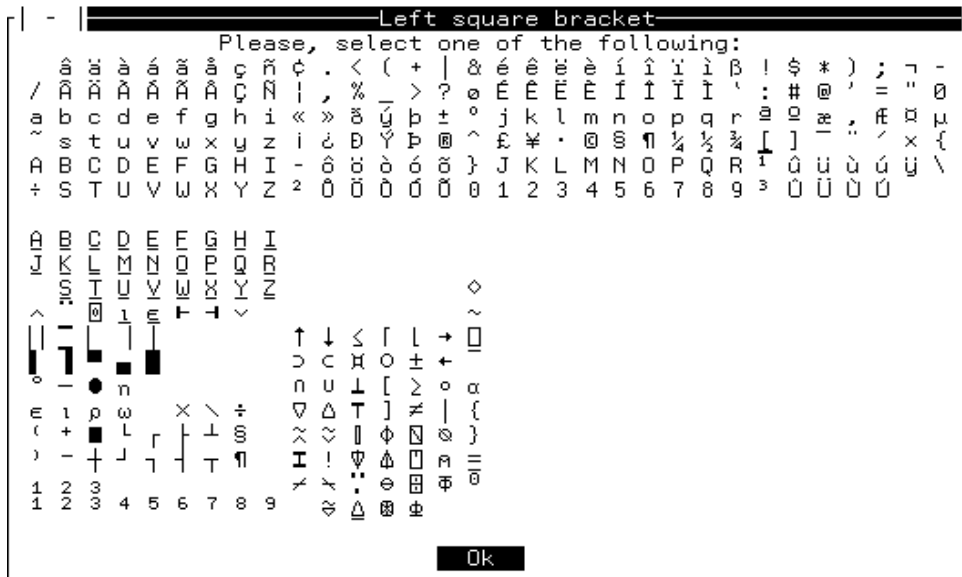
If you choose *Custom*, the Custom character set dialog box opens:



Initially, your cursor is on the input area for the left square bracket. You can type a value or press PF1 to select from a list. When you change the characters, decimal (dec) and hexadecimal (hex) values change automatically.

Try it now: To reset the characters to the original settings:

1. Check the *Reset* button.
2. When you are finished entering your customized characters, press PF4.
3. To display a screen with mapping information, press PF1.
4. When the Left square bracket dialog box opens, to select a value from this box, move your cursor to the appropriate character and press Enter.



Reference The Painter Dialog Boxes

To change any of your choices later in the session, you can use the Painter's File and Forms menus.

Dialog boxes request information about a task or supply information. After you specify options, you click a button to carry out an action.

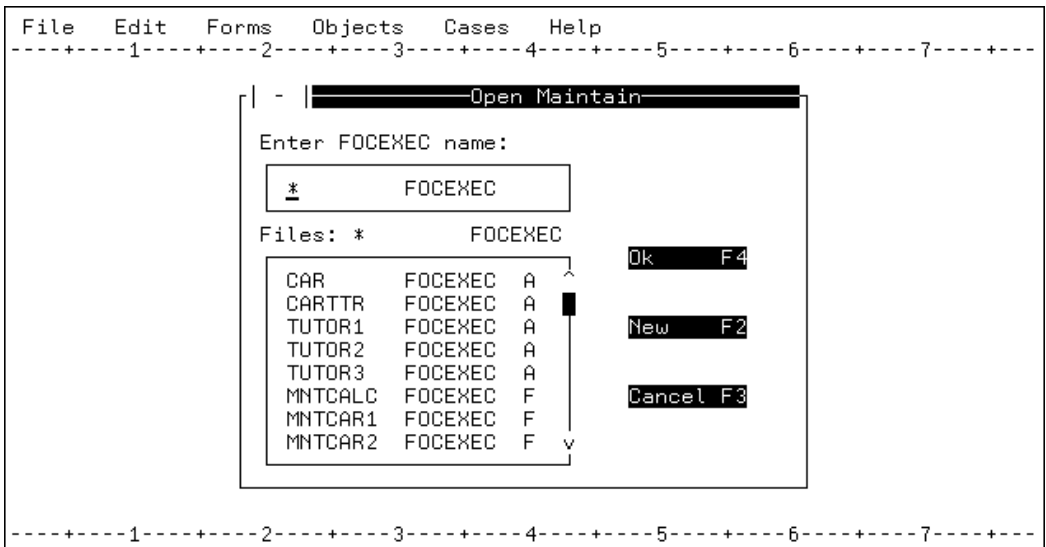
When the dialog box opens, the cursor is on the first option.

- To move to another option, press the Tab key to move forward, or press the Shift+Tab keys to move backward.
- The PF3 key usually cancels an action or signals No.
- The PF4 key usually signals OK or Yes.
- When a dialog box offering several options has a default option, pressing Enter always selects the default.

Naming the Procedure

A Maintain procedure that displays Winforms is composed of a pair of files: a Maintain procedure (either FOCEXEC or MAINTAIN) file and a WINFORMS file. Each pair has the same name (file name in CMS or member name in MVS): this is the name of the procedure. The Open Maintain dialog box asks you to enter this name. If the Maintain procedure and WINFORMS files do not exist, the Painter creates them.

The Open Maintain dialog box includes a combo box (a combination of the entry field and the list box) and command buttons.



Entry Field. An entry field is a field in which you type information. If a field permits a value longer than the field's width, you can scroll the field to enter more text. PF10 scrolls left and PF11 scrolls right.

List Box. A list box displays a list of choices. To select an item from the list, position the cursor on the desired item and press Enter.

If the list is longer than the box, you can use the scroll bars to move through the list.

To scroll:	Move the cursor to:
One line up or down	The up or down arrow on the vertical scroll bar and press Enter.
One screen up or down	Just above or below the arrow on the vertical scroll bar and press Enter.
One position left or right	The right or left arrow on the horizontal scroll bar and press Enter.
One screen left or right	Just to the right or left of the arrow on the horizontal scroll bar and press Enter.

When you scroll through a list, the scroll box moves up, down, left, or right to indicate where you are in the list.

Combo Box. When an entry field appears together with a list box, the combination is known as a combo box. You can enter a value by selecting it from the list box or by typing it directly into the entry field.

Note: The combo boxes that you create inside the painter (by selecting Combo box from the Objects menu) are different. In these, you cannot type directly into the entry field.

Command Buttons. Click a command button to initiate an immediate action, such as issuing or canceling a command.

You click a button by positioning the cursor on the button and pressing Enter. If the button has a shortcut key, it is faster to press the associated key. (A shortcut key is a function key assigned to a button. When a button has a shortcut key, the key name is usually displayed next to the button name.)

In the Open Maintain dialog box, you can name the procedure you wish to open by either:

- Typing the name of the Maintain procedure in the entry field and pressing Enter (or moving the cursor to the *OK* button and pressing Enter).

Adding FOCEXEC is optional.

Pressing F2 also works if the Maintain procedure does not already exist. If it exists and you press F2, you are asked if you want to use the existing Maintain procedure.

Step 1: Creating a New Winform

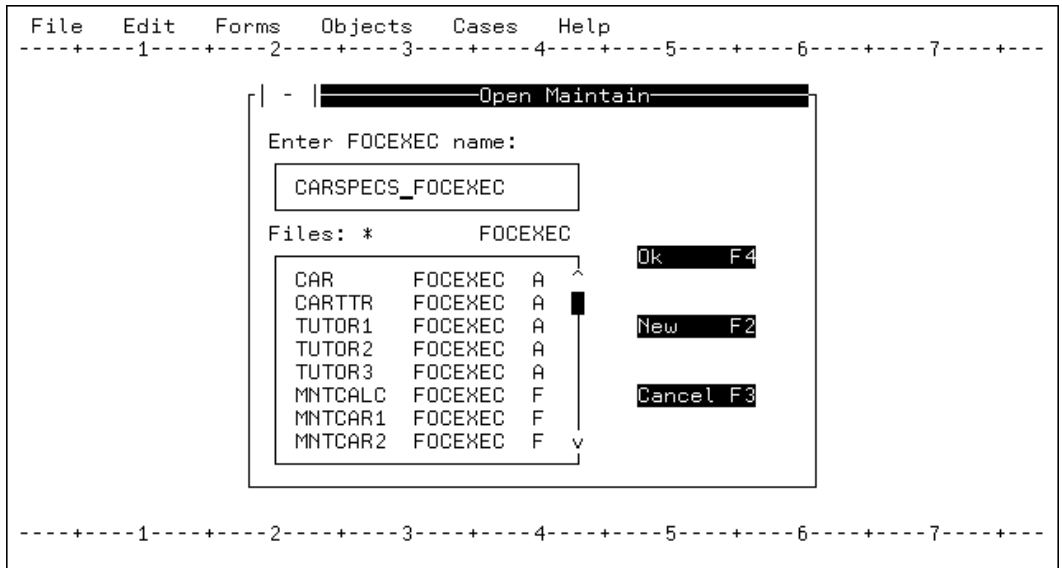
- Pressing Tab to position the cursor in the list box, then moving the cursor to the desired procedure and pressing Enter.

If the list of Maintain procedures is too long to fit in the box, you can use search criteria to narrow the list. For example, if you want to see Maintain procedures that start with the letter C, enter C*.

If you do not see the Maintain procedure, you can press PF8 to scroll forward, PF7 to scroll backward, or use the scroll bars. As you scroll through the list, notice that the scroll box indicator moves to show where you are in the list.

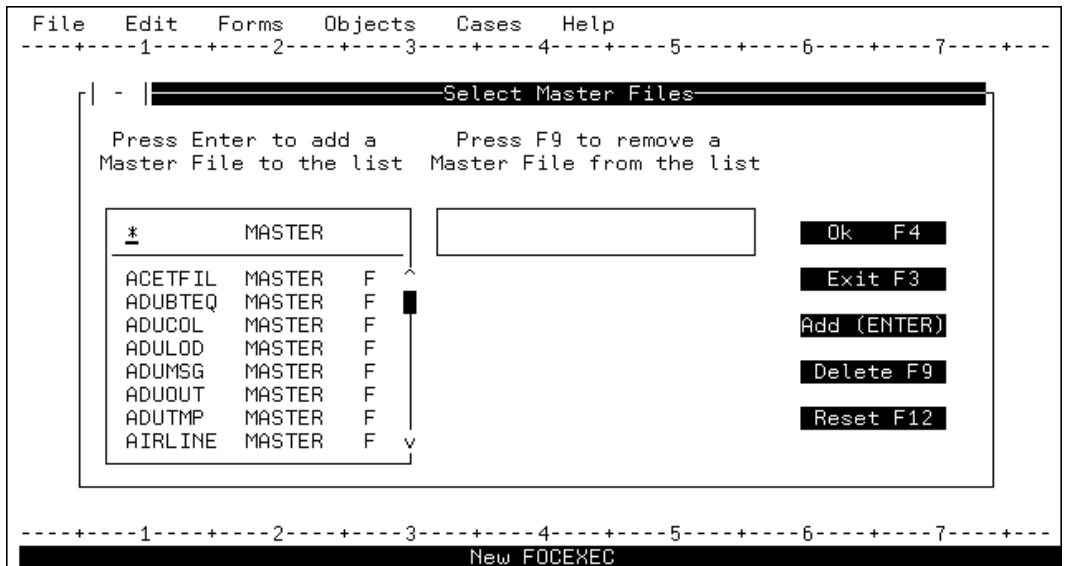
If you press PF3 (or click the *Cancel* button), you exit the Painter.

Try it now: Type *CARSPECS* in the entry field and press Enter (or move the cursor to the *OK* button and press Enter) as you will be creating a new procedure named *CARSPECS*.



Selecting Master Files

After you enter the name of the Maintain procedure, the Select Master Files dialog box appears. This is where you select the data sources you access in this Winform. (This is the same dialog box that appears if you chose *Select Master* from the File menu.) For the purposes of this tutorial, you entered a name of a Maintain procedure that does not exist. When the Maintain procedure exists, you are not prompted for the data source(s).



When the dialog box first opens, your cursor is positioned so you can type the name of the file. You can either type the name (the word MASTER is optional), or you can select from the list box. After you press Enter, the data source name appears in the box to the right of the list of available Master Files. You may select up to 16 data sources.

The data sources selected determine which fields are displayed in the Field dialog box that is used to add fields to the form.

If you add a data source you do not want, you can remove it by moving the cursor to the data source name and pressing PF9. Note that PF9 does not delete the name from the list. To remove several data sources, you can press PF12 (or click the Reset button) to clear all changes.

If your procedure will not access a data source (for example, a menu), you need not select one. In that case, when the dialog box opens, press PF4 or the *OK* button. If you press PF3 (or click the *Exit* button), you return to the Open Maintain dialog box.

When you have finished adding data sources, press PF4 or move the cursor to the *OK* button and press Enter.

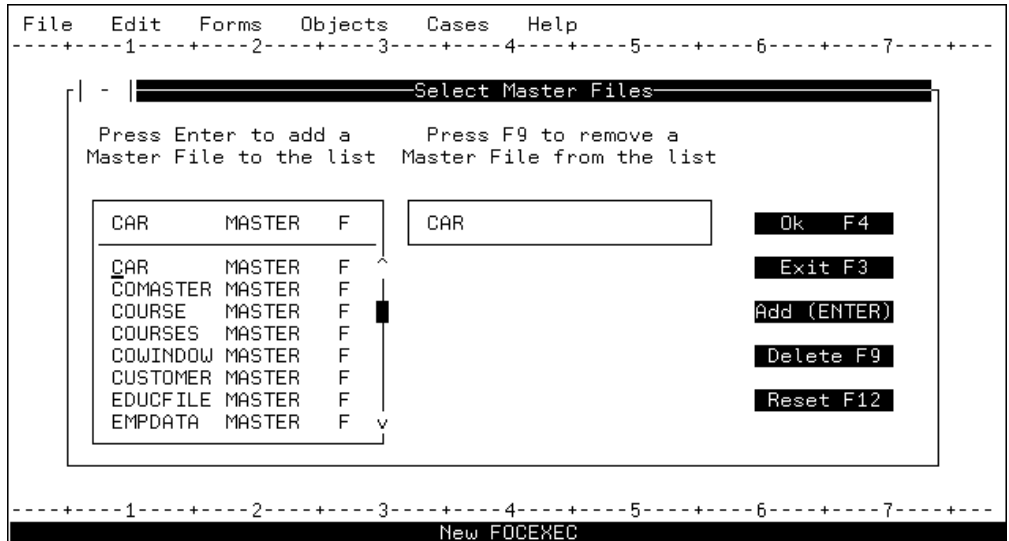
Step 1: Creating a New Winform

For the purposes of this tutorial, you access the Car sample data source.

Try it now:

1. Select **CAR** from the combo box on the left.

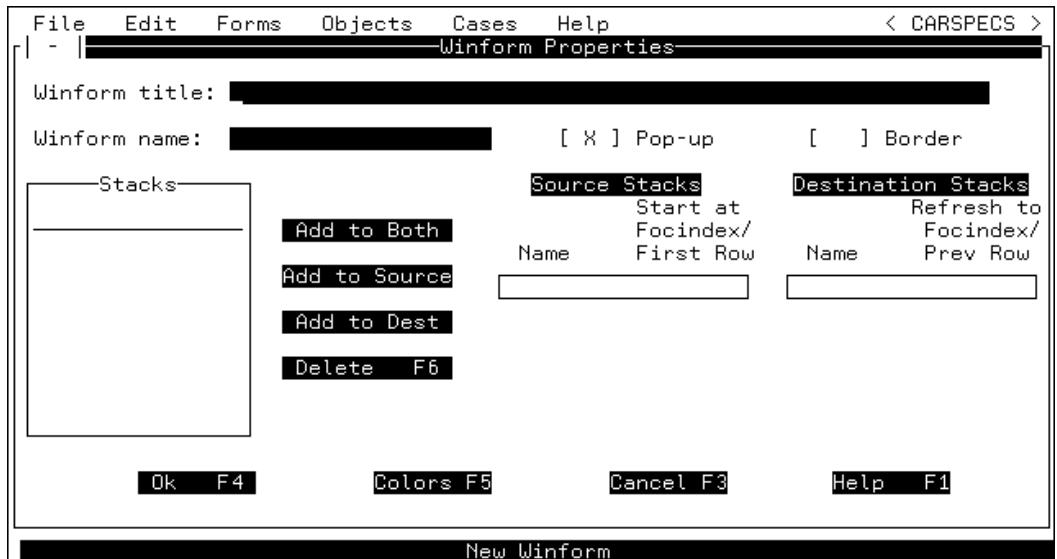
Your screen should look similar to the following.



2. Press PF4 to indicate that you have finished or move the cursor to the **OK** button and press Enter.

Defining the Winform's Properties

The Winform Properties dialog box opens and requests information about the form. It looks similar to the following.



Winform Title

The Winform title is optional. The title is centered in the Winform's top border. When a Winform has no border, it does not display a title.

Try it now: Type *Car Editing Form* as the title of this Winform and tab to the Winform name field.

Winform Name

Winform Name is the name used to identify the Winform to FOCUS. You refer to the Winform by its name in the WINFORM command and in the Winform Painter.

The Winform name is required.

Try it now: Type *ShowCars* as the Winform name.

Pop-up Check Box

A check box is a switch that lets you select or deselect an option. Square brackets or angle brackets are used to indicate a check box (depending on your terminal emulation and on how your Painter session is configured in the Terminal suboption in Preferences of the File menu). You select a check box by typing an "X" or other character into it.

If a Winform is a pop-up, it disappears when it is closed (by a WINFORM CLOSE command). By default, the Pop-up check box is selected.

Try it now: Keep the default setting as the tutorial requires this Winform to be a pop-up.

Border Check Box

When you select this attribute, it displays a box around the perimeter of the Winform. It is optional, but must be selected to display a Winform title or a Control menu.

Try it now: Select the *Border* check box, as this tutorial requires a border.

Stacks

A stack is a simple table. Every stack column corresponds to a data source or user-defined variable. Every stack row corresponds to a data source record (a path instance). The stack itself represents a data source path. You can populate a stack by retrieving data from a data source, calculating values, or copying all or part of an existing stack.

The Current Area is Maintain's unnamed default stack and has one row.

Winforms do not display data directly from a data source and do not directly update a data source. They display data from, and write data to, stacks or the Current Area. These are known as the source and destination stacks. For each Winform, you can use as many source and destination stacks as you wish.

You can select any of the stacks in the Stacks list box as a source or destination stack for every field, browser, and grid that you create in all the Winforms in the Winform file. You select the desired stack and then click the appropriate Add button to the right of the list. Depending on the button you clicked, the stack name is copied into the Source Stacks and/or Destination Stacks list boxes at the right of the dialog box.

The source and destination stacks always must be identical or none.

If you do not specify any source stacks, the source defaults to the Current Area. Similarly, if you do not specify any destination stacks, the destination defaults to the Current Area.

Since this is a new Winform file, the Stacks list is empty. You can add stacks to this list by typing the desired stack name in the entry field (above the line in the box), and then clicking the appropriate Add button. The stack name is copied to the Stacks list box, as well as to the Source Stacks and/or Destination Stacks list boxes.

Try it now: Type *CarStack* in the Stacks entry field and then press Enter (which clicks the default *Add to Both* button). Next, backspace over your first entry, type *BodyStack*, and press Enter.

CarStack and *BodyStack* appear in the Stacks, Source Stacks, and Destination Stacks list boxes.

Source Stacks

For each stack in the Source Stacks list, you can check the *Start at FocIndex* check box. This determines the current row (the current position within the stack) when the Winform opens.

If you design the Winform with *Start at FocIndex* checked, when the Winform opens, the stack starts out with the same position it had just prior to opening the Winform.

This ensures that the stack's position is consistent inside and outside the Winform. Maintain accomplishes this by using the system variable *FocIndex* to determine the current row.

This enables you to retain the stack's position when you open the Winform and makes it possible for you to dynamically manipulate the current row by assigning a value to *FocIndex*.

If you design the Winform without *Start at FocIndex* checked, when the Winform opens, the stack's current position is the first row, regardless of where it was prior to opening the Winform.

Other aspects of *FocIndex* are described in Chapter 2, *Maintain Concepts*.

Try it now: For all of this application's stacks, check *Start at FocIndex* (the default).

Destination Stacks

For each stack in the Destination Stacks list box, you can check the *Refresh to FocIndex* check box. This controls the Winform's behavior when a user invokes a trigger that interrupts—and later returns control to—the Winform. When control returns to the Winform, the data that it displayed previously is refreshed (in case the stack had been updated in the interim).

If you design the Winform with *Refresh to FocIndex* checked, when the Winform refreshes its data from the stack, it also refreshes its position within the stack.

This ensures that the Winform reflects the most recent changes not only to the stack's data, but also to its position. It accomplishes this using the system variable, *FocIndex*, to determine the current stack row.

When a trigger manipulates the stack and changes the current position, for example, if the trigger calls a second Winform (that also displays that stack), and a user moves to another row, *Maintain* retains that new stack position when the user returns to the original Winform.

This also enables you to dynamically manipulate the current row by assigning a value to `FocIndex` within the intervening trigger.

If you design the Winform without `Refresh to FocIndex` checked, the current row is unchanged by anything that happened during the trigger.

Other aspects of `FocIndex` are described in Chapter 2, *Maintain Concepts*.

For all of this application's stacks, check `Refresh to FocIndex` (the default).

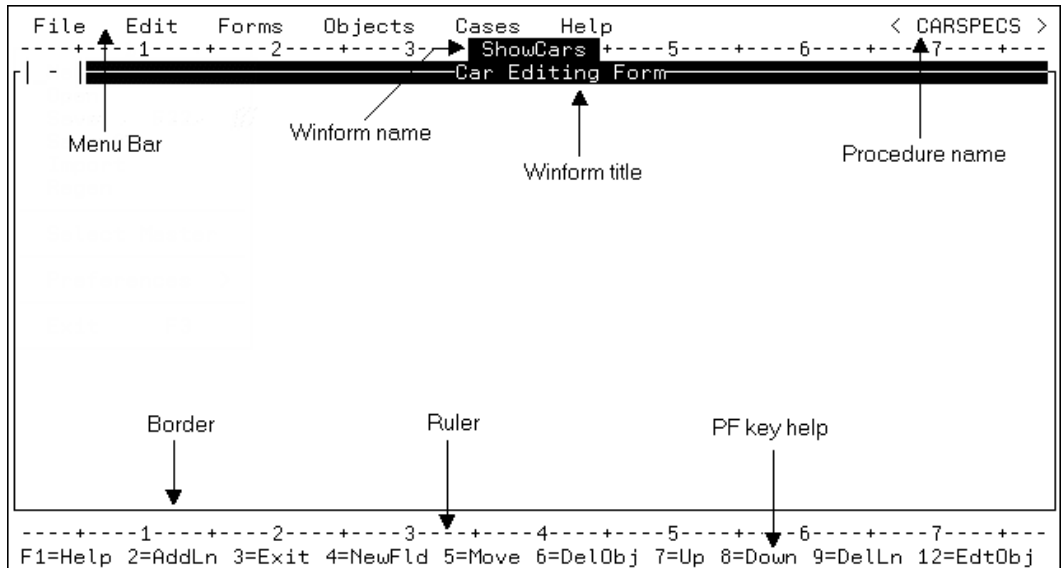
When you have finished completing the form, the screen should look similar to the following:



Try it now: When you are done, click the *OK* button.

You have completed the Painter's preliminary dialog boxes. After you click the *OK* button, the Painter displays the Design Screen where you can paint your Winform.

The following diagram identifies the different parts of the Winform Design Screen:



Using the Painter's Menus

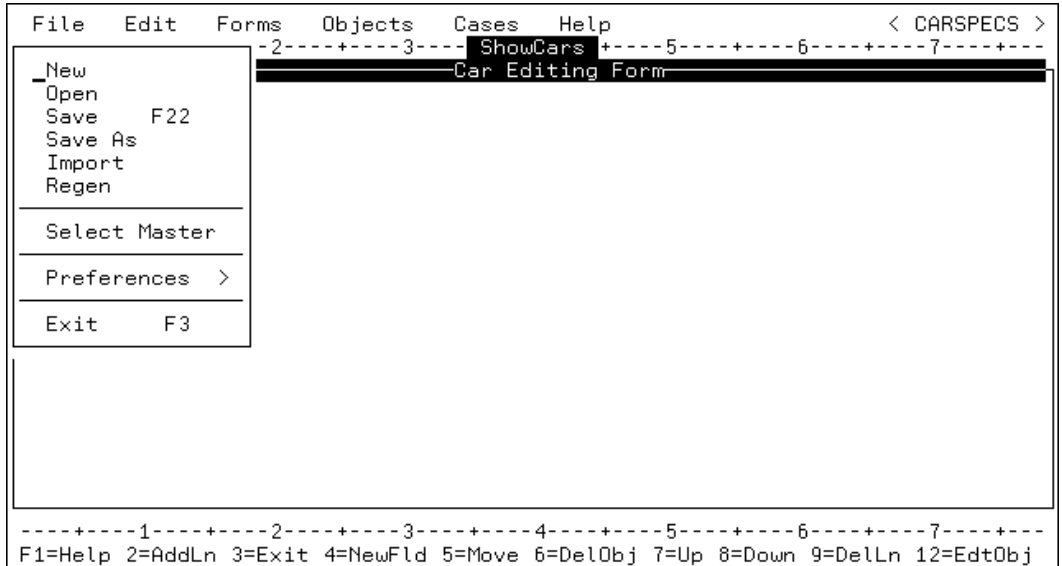
To view the menus available to you in the Maintain Winform Painter, display the pull-down menus. To select a menu from the menu bar you can press:

Function keys. Press PF10 or the Home key to position the cursor on the space immediately to the left of the first menu. You can press Tab or PF11 to move forward through the menu bar. Press Shift+Tab to move backward through the menu bar. When you are on the desired menu, press Enter.

Arrow keys. Use the arrow keys to move the cursor to the desired menu and then press Enter.

Once you have opened a menu, you can move through the menu options using Tab or the arrow keys. To select the desired menu option, position the cursor on the option's line and press Enter.

Try it now: Press PF10 and then Enter to display the File pull-down menu:



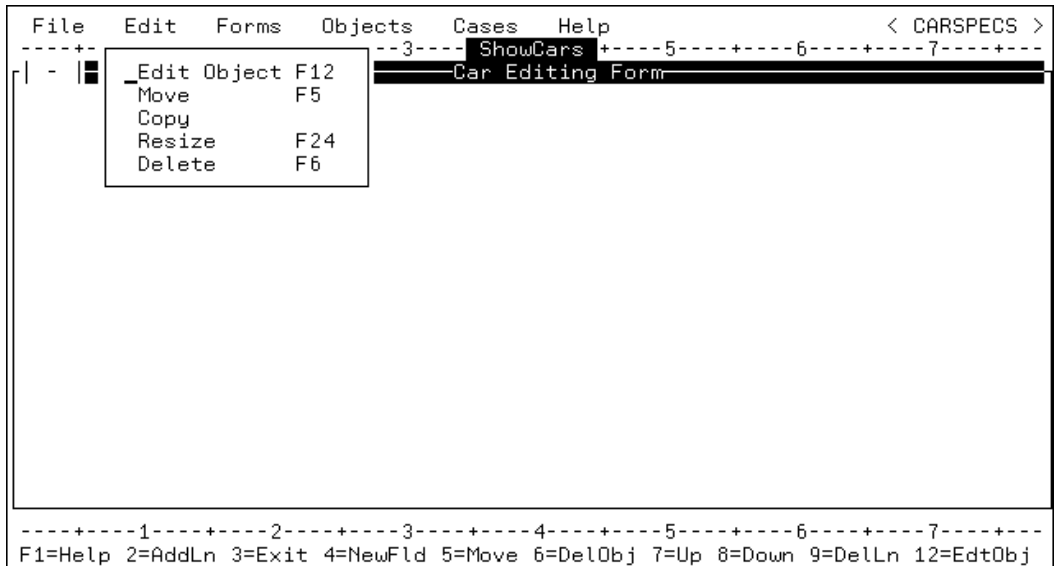
Press PF3 to close a pull-down menu. In most Painter contexts, PF3 cancels the current activity.

Note: While the cursor is positioned on an empty part of the design screen (not on a menu or the menu bar), pressing PF3 quits the Painter session (after asking you to confirm that you want to quit).

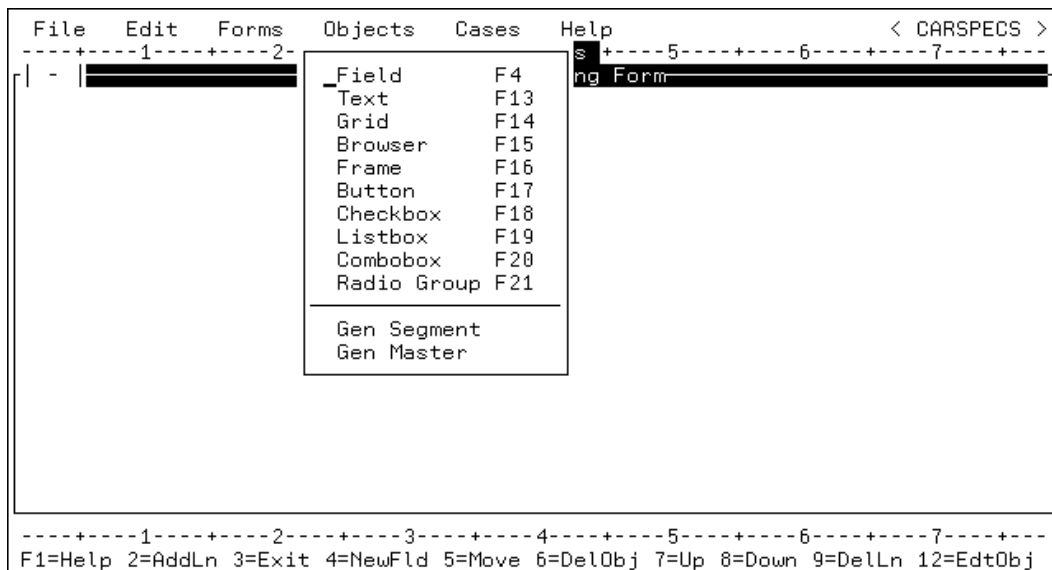
If you press PF3 by mistake, press PF3 again to cancel the quit operation.

As you cycle through the menus, notice that some of the pull-down menu items have a function key displayed to the right of the item name. This identifies the key as the shortcut key for the menu item. For example, in the Objects menu, F4 appears to the right of the Field option. This indicates that you can press PF4 to select *Objects* and *Field* from the menu bar. Using the shortcut keys saves menu navigation (and therefore saves time).

Try it now: Display the Edit, Forms, and Objects menus to become familiar with the Painter's functions.



Defining the Winform's Properties



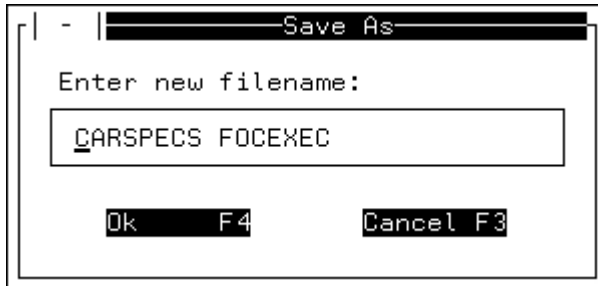
Saving Your Work and Exiting

It is a good idea to save your work from time to time. This way, if a system problem interrupts your Painter session or you accidentally delete some work, you can restart your Painter session from the point of your last save.

Saving the Procedure

Try it now: Move the cursor to the File pull-down menu by pressing the Home key or PF10. Select either *Save* or *Save As* from the menu by moving the cursor to the appropriate item and pressing the Enter key.

- To save your Maintain procedure and WINFORMS files under their current name (CARSPECS), move the cursor to *Save* and press Enter or alternatively, press PF22 (the Save shortcut key). The keyboard locks while the file is being saved. When it unlocks, you can continue painting the screen if you wish.
- To save your Maintain procedure and WINFORMS files under a different name, select *Save As*. When the Save As dialog box opens, the Filename field displays the current Maintain procedure name. After you type the new name, press the Enter key to save the file.



Exiting the Winform Painter

There are two ways to exit from the Winform Painter: select *Exit* from the File Menu or press PF3.

Pressing PF3 exits whatever you are doing. For example, if a dialog box or a pull-down menu is open, the dialog box or menu disappears.

If you are not selecting from a dialog box or menu, pressing PF3 exits the Painter.

If you made any changes since the last time you saved the file, you are asked if you want to save the changes. Move the cursor to the appropriate button and press the Enter key.



Try it now: If you wish to continue the tutorial at this point, begin Step 2 in the next section. If you wish to exit now, press PF3. You can continue later where you left off as explained when you begin *Step 2: Adding Fields* on page 4-20.

Step 2: Adding Fields

If you exited from the Winform Painter, you can display it again by:

- Entering MPAINT at the FOCUS command prompt. When the Open Maintain dialog box appears, type or select CARSPECS and press Enter.
- Entering MPAINT CARSPECS at the FOCUS command prompt.

In either case, you return to the Painter exactly where you left off.

Adding the First Field

Perform one of the following to add a field to the form:

- Select *Field* from the Objects pull-down menu. At the bottom of the screen, the PF key legend disappears. In its place, a message appears telling you to move the cursor to where you want to position the field. The cursor is positioned where it was when you last pressed Enter. You must move the cursor to the starting position for the field and press Enter.
- Position the cursor at the screen position where you want the field to begin and press PF4. (PF4 is the shortcut key for adding a field.)

If your terminal emulator indicates the cursor's row and column position, position the beginning of the field at row 6, column 27. This tutorial uses emulator coordinates for positioning controls in the Winform. (The Winform Painter refers to controls as objects.)

To determine the same position using Winform row and column equivalents, subtract 3 from the emulator row position, and 1 from the emulator column position. For example, if you are determining row position by manually counting from the first, that is, the top line inside the Winform, this is row 3 and if you are determining column position by using the rulers above and below the Winform, this is column 26.

The Painter displays the following dialog box, and the cursor flashes in the Field entry field:

The dialog box is titled "Field" and is part of the "Car Editing Form" application. It contains the following fields and options:

- Field List:** COUNTRY, CAR, MODEL, BODYTYPE, SEATS, DEALER_COST, RETAIL_COST, SALES, LENGTH, WIDTH.
- Source:**
 - (*) Default
 - () Current Area
 - () Stack
- Destination:**
 - Destination Stack
 - (None)
- Default value:** []
- Length =** []
- Prompt:** [X]
- Uppercase:** [X]
- Protected:** []
- Object Name:** Field1
- Buttons:** Ok (F4), Accepts (F6), Triggers (F12), Color (F5), Cancel (F3).

You can specify the following in this dialog box:

- The name of the field.
- The source and destination of field's data.
- Whether the text the user enters should be converted to uppercase.
- The length of the field.
- Whether the field allows data entry or is protected.
- Whether the field name should be used as a prompt.
- The values the field should accept.
- The name by which you can refer to the Winform field in your procedure logic.
- The color of the field.
- The triggers for the field.

Field

To specify the field to display, type a name in the Field entry field and then press Enter; or scroll through the list of field names in the Field list box, move the cursor to the appropriate field name, and press Enter.

Try it now: For the purposes of this tutorial, specify the *Country* field.

Object Name

All controls—except frames and browsers—have names. The Winform Painter refers to controls as objects, and to their names as object names. These names enable you to refer to controls within your procedure, so that you can dynamically manipulate them in response to run-time events.

Object Name is located in the bottom middle of the dialog box.

Try it now: Name this field *CountryField*, by overwriting the default name (*Field1*).

Source

The Source radio button group enables you to select a source for the field's initial value:

- **Default Value.** Check this source to specify a constant that is always displayed for this field when the form initially opens, for example, a year field with a default value of the current year. Enter a default value if the default display for the user should come from a constant. The Winform Painter does not support default values for fields in the current release.
- **Current Area.** Check this source if the data to be displayed is to come from the Current Area variable with the same name as the field specified. The Current Area is essentially a stack that has one row.
- **Stack**

The Source Stack drop box lists every stack that was specified in the Winform Properties dialog box.

If no stacks appear, exit this screen and return to the Form Properties dialog box to enter stacks.

When you click the Stacks radio button and then click the down-arrow button, the Painter displays a list of available stacks.

To click a radio button, position your cursor in the middle of the desired button and press any character, for example, an "x".

To select one of these stacks as the source of the field's initial value, position the cursor on the desired stack and press Enter.

Try it now: Select *CarStack* as your source. In this tutorial, you obtain your input data from this stack.

When you press Enter or a function key—such as PF4 to confirm your entries and close the dialog box—the character you entered changes into an asterisk (*).

Destination

Note that the Source and Destination must be the same. Both must either display <None> or the name of the stack.

The destination specifies where the field value supplied by the end-user is to go. The Destination Stack drop box lists every stack that was specified in the Winform Properties dialog box.

When you click the down-arrow button, the Painter displays a list of available stacks.

To select a stack as the destination, position the cursor on the desired stack and press Enter.

If you want the field's value to be written to the Current Area, do not specify anything. Whether or not you select a destination stack, all values are automatically copied to the Current Area.

Try it now: For this tutorial, specify *CarStack* as the destination stack.

Length

The Painter automatically fills this field when you select a field from the Field list box or when you enter a data source field name in the Field entry field. In this case, the Painter fills in the length the next time you press Enter.

The Length is obtained from the Master File.

You can override the default length by typing in a different length.

If the field length is longer than what is specified in the Master File, all of the characters specified after the end of the Master length are not added to the data source.

Try it now: For this tutorial you want the length specified in the Master File, so you need not change the specification.

Prompt

The Prompt check box indicates whether or not you want the field name (or other text) to appear to the left of the field.

You can change the value in the Prompt entry field. However, the prompt text may not exceed 13 characters. If you want the prompt text to be longer than 13 characters, you must uncheck *Prompt* and supply your own text using the method described in *Step 4: Adding Text* on page 4-33.

Try it now: For the Country field, you want the field name to appear, so accept the default (checked). To enhance the prompt's appearance, change all the letters following the first to lowercase, so that it reads *Country*.

Uppercase

When the Uppercase check box is checked (this is the default setting), whatever text the user types in is converted to uppercase before being placed in the destination.

Try it now: Accept the default. For this tutorial, all data should be converted to uppercase, because all data in the Car data source is stored in uppercase.

Protected

Select the Protected check box to protect the field from users changing its value. You check this box if this field is to be display-only.

Try it now: In general, you want to protect key fields, such as Country, so check this option.

OK (PF4)

Click *OK* (or PF4) after you finish making all changes to the field dialog box. If you want to view the dialog boxes associated with clicking PF5 and PF6, do not click PF4 yet.

Accepts (PF6)

The Accepts button opens the Accept List dialog box which enables you to specify a data validation test for the field, as well as the conditions under which to display the list or range of valid values. If valid values are already specified for this field in the Master File, you use this dialog box to specify when to display them.

For this tutorial you do not use the Accepts data validation feature. Click PF3 to exit from this dialog box.

Color (PF5)

Click the Color button when you want to change the colors for a field. When you click this button, the following dialog box opens:

```

| - | Set Colors
|   |
|   | Color
|   | ( * ) Default
|   | (   ) Green
|   | (   ) Blue
|   | (   ) Red
|   | (   ) Pink
|   | (   ) Turquoise
|   | (   ) Yellow
|   | (   ) White
|   |
|   | Attr
|   | ( * ) None
|   | (   ) Highlight
|   | (   ) Nondisplay
|   |
|   | Ext Attr
|   | ( * ) Normal
|   | (   ) Blink
|   | (   ) Reverse
|   | (   ) Underline
|   |
|   | Ok F4      Cancel F3
  
```

To select a color, an attribute, or an extended attribute, move the cursor to the appropriate radio button and press any key.

Try it now: For this tutorial, you do not show colors. Click PF3 to exit from this dialog box.

Cancel (PF3)

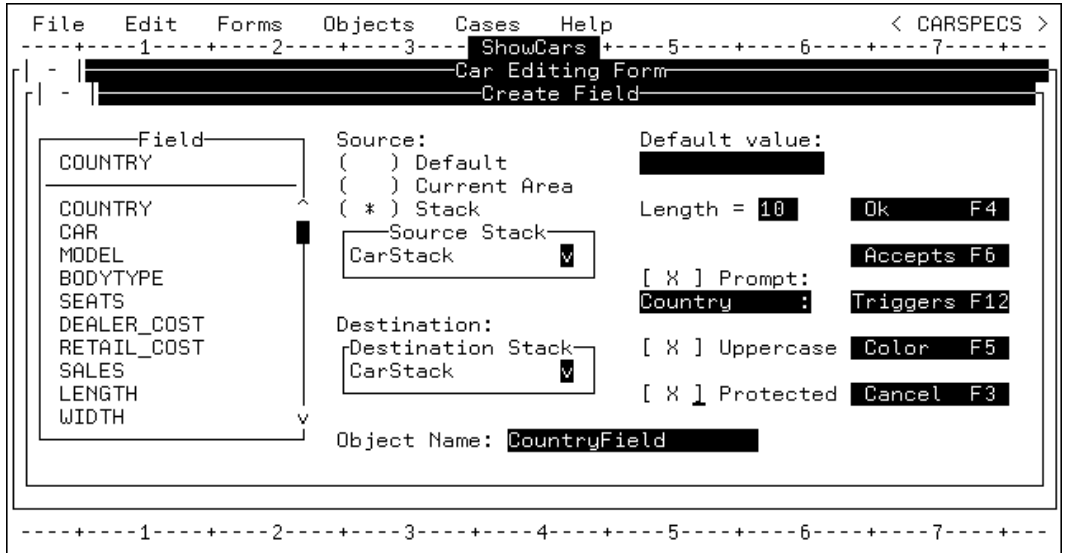
When you click the Cancel button, any changes made to the Field dialog box are ignored, and you return to the Winform Design Screen.

Triggers (PF12)

When you click the Triggers button, the Triggers dialog box opens. It lists all of the system shortcut keys and enables you to enter functions (which are referred to as cases in the Winform Painter) against the corresponding keys.

Step 2: Adding Fields

When you finish entering options, the Field dialog box should look similar to the following:



Try it now: Press PF4 to indicate that you are done.

After you press PF4, the Country field appears, and the form looks similar to the following:



Adding Additional Fields

To add additional fields, move the cursor two lines below the Country field and press PF4. The Field dialog box appears.

Note that the changes you specified in the Field dialog box for the Country field (for example, the CarStack stack) are the new defaults.

Try it now: Select the *Car* field.

The length is obtained from the Master File.

1. Ensure that the *Prompt*, *Uppercase*, and *Protected* check boxes are checked.
2. Change the case of the prompt so that it reads *Car*.
3. Before pressing PF4 to indicate that you are finished entering specifications, check the dialog box for the Car field to confirm that it looks like the following:

The screenshot shows a 'Create Field' dialog box with the following details:

- Field List:** CAR (selected), COUNTRY, CAR, MODEL, BODYTYPE, SEATS, DEALER_COST, RETAIL_COST, SALES, LENGTH, WIDTH.
- Source:** () Default, () Current Area, (*) Stack. Source Stack: CarStack.
- Destination:** Destination Stack: CarStack.
- Object Name:** Field2
- Default value:** [Blank]
- Length:** 16
- Triggers:** [X] Prompt: Car; [X] Uppercase; [X] Protected.
- Buttons:** Ok (F4), Accepts (F6), Triggers (F12), Color (F5), Cancel (F3).

4. Press PF4 when you are finished entering specifications for the Car field.

Editing, Moving, and Resizing Controls

When you are adding fields (or other controls later in the tutorial), there are two ways to make changes. (Winform Painter refers to controls as objects.)

One way is to remove the control and begin again. To remove a control:

- Select *Delete* from the Edit pull-down menu. Move the cursor to the control you want to delete and press Enter.
- Alternatively, move the cursor to the control you want to delete and press PF6.

Step 2: Adding Fields

The control no longer appears on the screen. Editing the control presents an easier way to make changes. To edit a control:

- Select *Edit Object* from the Edit pull-down menu. Move the cursor to the control you want to change and press Enter.
- Alternatively, move the cursor to the control you want to edit and press PF12.

The appropriate dialog box opens, and you can specify changes.

You may wish to move controls around on the screen to make the screen more appealing. To move a control:

- Select *Move* from the Edit menu. Position the cursor on the control you want to move and press Enter. Next, position the cursor to where you want the upper left corner of the control to be and press Enter.
- Alternatively, position the cursor at the control you want to move and press PF5. Next, position the cursor where you want the upper left corner of the control to be and press Enter.

You may want to resize a control, such as the grid described in the next section, *Step 3: Adding a Grid* on page 4-29. To change the size of a control:

- Select *Resize* from the Edit menu. Move the cursor to the control you want to resize and press Enter. The border of the control is highlighted and the cursor is at the bottom right of the control. Move your cursor to where you want the bottom right corner to be and press Enter.
- Alternatively, move the cursor to the control you want to resize and press PF24. The border changes as described in the previous paragraph. Move the cursor to indicate the new size of the control and press Enter.

Step 3: Adding a Grid

A grid is a stack editor that enables you to display and change the values in a stack. The columns are listed across the grid, and one or more rows appear at one time. In this section, you add a grid that displays BodyStack. A sample grid appears that looks similar to the following:

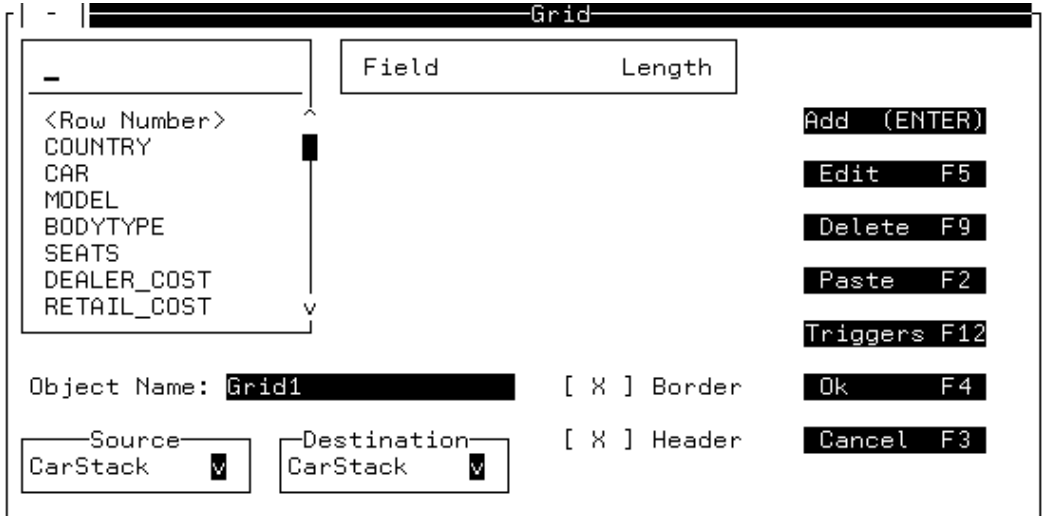
The screenshot shows a terminal window titled "Car Editing Form". At the top, it displays "Country : ENGLAND" and "Car : JAGUAR". Below this, a box labeled "Grid" contains a table with three columns: "MODEL", "BODYTYPE", and "DEALER_COST". The table lists two car models: "1 V12XKE AUTO" (Convertible, 7,427) and "2 XJ12L AUTO" (Sedan, 11,194). Below the grid, a prompt reads "Select country and car, then update old cost data". At the bottom, there are four function key buttons: "Previous Car F1", "Next Car F2", "Quit F3", and "Done F4".

MODEL	BODYTYPE	DEALER_COST
1 V12XKE AUTO	CONVERTIBLE	7,427
2 XJ12L AUTO	SEDAN	11,194

Try it now: To create a grid, select *Grid* from the Objects menu. When the Painter asks you to define the grid's upper-left and lower-right corners, position the upper-left corner of the grid in row 11, column 6 (using the terminal emulator's coordinates) and place the lower-right corner in row 16, column 76. These specifications allow sufficient space for the column headings and data.

Step 3: Adding a Grid

After you specify the coordinates, the following dialog box appears:



Adding Columns

From the combo box, you can select the fields you want to include as columns in the grid. The list box displays all fields from the Master Files that you specified in the Winform Properties dialog box when you began the tutorial.

To select fields:

- Type the field name in the Field entry field. (The cursor flashes in the entry field when the dialog box opens).
- Alternatively, move the cursor to the desired field and press Enter. If you do not see the field in the Field list box, move the cursor into the box and press PF7 or PF8 to scroll the box (or use the scroll bar).

In either case, after pressing Enter, the field you selected and the length from the Master File appear in the Grid Field List.

Try it now: For this tutorial, select the following fields:

- <Row Number>. This is a system-generated column that displays the sequence number of each row. It enables the user to identify the current position in the stack. When selected, it becomes the first grid column.
- Model.
- BodyType.
- Dealer_Cost.

- Retail_Cost.

As you select fields, the list of selected fields grows. You can edit this list in several ways:

- You can change the default field length (that is, column width) by typing over the length. The default is obtained from the Master File.
- You can delete columns by moving the cursor over a particular column name and pressing PF9. Another way to do this is to move the cursor to the column name, press Enter, and click the *Delete* button.
- You can move columns around by using a combination of Delete (PF9) and Paste (PF2). To move a column, first delete it as described in the previous paragraph. Next, move the cursor to the row preceding the desired position. Press PF2. The deleted column is inserted in the position you specified.

Changing Stacks

Below the list of available fields are two boxes indicating the grid's source and destination stacks. You specify BodyStack as both the source and the destination.

Try it now: In each box, click the down-arrow button. When the Painter displays a list of available stacks, select *BodyStack*.

After adding the five fields and changing the stacks, the dialog box looks similar to the following:

The screenshot shows a dialog box titled "Grid" with the following components:

- Field List:** A list of fields with their corresponding lengths:

Field	Length
<Row Number>	2
MODEL	24
BODYTYPE	12
DEALER_COST	9
RETAIL_COST	9
- Available Fields:** A list of fields including RETAIL_COST, <Row Number>, COUNTRY, CAR, MODEL, BODYTYPE, SEATS, DEALER_COST, and RETAIL_COST.
- Object Name:** Grid1
- Source Stack:** BodyStack
- Destination Stack:** BodyStack
- Buttons:** Add (ENTER), Edit F5, Delete F9, Paste F2, Triggers F12, Ok F4, and Cancel F3.
- Options:** [X] Border and [X] Header.

Protecting and Unprotecting Columns

Because you previously protected the Country and Car fields from editing, the current default is Protected. Leave the Model and BodyType columns protected, because they are key fields in the Master File, but unprotect the Dealer_Cost and Retail_Cost fields.

Try it now: To protect or unprotect a column, move the cursor to the column (in this case, *Dealer_Cost*) and press PF5 to edit.

The Painter displays the following dialog box:

The screenshot shows a dialog box titled "Edit Column". On the left, there is a list box labeled "Field" containing the following items: DEALER_COST, COUNTRY, CAR, MODEL, BODYTYPE, SEATS, DEALER_COST, RETAIL_COST, and SALES. A vertical scrollbar is on the right of this list. On the right side of the dialog, there are several fields: "Title:" with the value "DEALER_COST", "Length:" with the value "9", and two checkboxes: "[X] Uppercase" and "[X] Protected". At the bottom of the dialog, there are four buttons: "Ok F4", "Accepts F5", "Color F6", and "Cancel F3".

1. Uncheck the *Protected* check box and click *OK* to confirm.
2. Repeat the previous process to unprotect the *Retail_Cost* column. You may find the column already unprotected if you created it after you changed the default for *Dealer_Cost*.
3. Click *OK* to confirm the change to *Retail_Cost*.

In addition to protecting and unprotecting columns, you can edit other column attributes in the *Edit Column* dialog box, including length, title, and case-sensitivity.

Color and data validation (*Accepts*) are not supported for grids in the current release.

4. Click *OK* in the *Grid* dialog box to confirm the grid's design.

After closing the dialog box, the screen looks similar to the following:

Note: The word *BodyStack* is centered at the top of the box around the grid. This name only appears during the painting process so that the developer can quickly identify which stack is used in that grid. It does not appear when the application is executed.

Step 4: Adding Text

There are two ways of adding text to the screen:

Create a text control. The Winform facility enables you to treat text like other types of controls, such as buttons, grids, and fields. You can define a message or other text as a text control. Like other types of controls, you manipulate a text control using the Painter menus and function keys (not the Delete and Insert keys).

- You can assign a color.
- You can move the text as a single unit.
- You can delete the text as a single unit.

To create a text control, select *Text* from the Objects menu or press PF13. The cursor flashes where it was the last time you pressed Enter. You are prompted to move the cursor to where you want the text to begin.

Step 4: Adding Text

After pressing Enter, the Create Text dialog box appears, and you can type the desired text in the entry field. You can include any characters you wish, with the exception of double quotation marks ("). When you are finished, you click PF4, and the text appears on the screen. If the text is wider than the window, scroll left and right by pressing PF10 and PF11.

Copy a text control. You can copy text controls from within the Painter. This feature is useful when you want several text controls to look similar to one another.

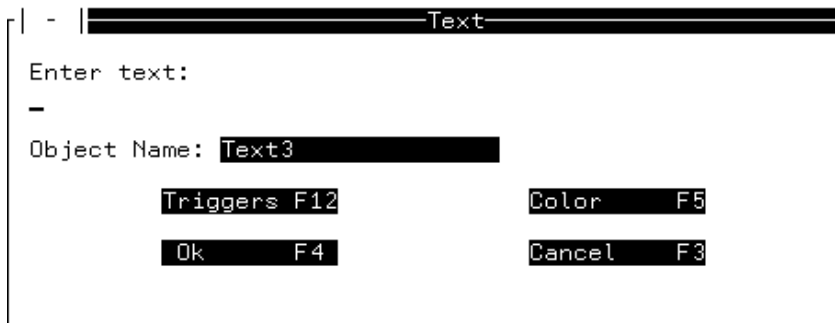
To copy a text control, select *Copy* from the Edit menu. Next, move your cursor to the control you want to copy. Then, move your cursor to where you want the copied control to be located. You can edit the control by pressing PF12 to change its attributes.

For this tutorial, you want to display a message telling application users how to use the Winform.

Try it now:

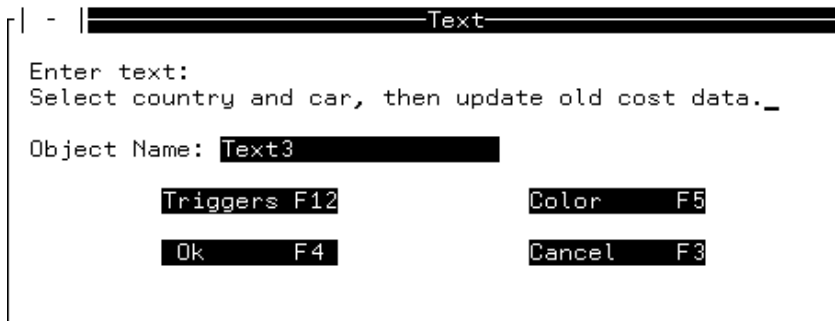
1. Select *Text* from the Objects menu. Move the cursor to row 18, column 15 (using the terminal emulator's coordinates) and press Enter.

A dialog box similar to the following opens.



```
| - |-----Text-----|
Enter text:
-
Object Name: Text3
Triggers F12          Color F5
Ok F4                Cancel F3
```

2. In *Enter text;* type the following: *Select country and car, then update old cost data.*



```
| - |-----Text-----|
Enter text:
Select country and car, then update old cost data._
Object Name: Text3
Triggers F12          Color F5
Ok F4                Cancel F3
```

- Press PF4 to indicate you are finished.

The screen should look similar to the following:

```

File  Edit  Forms  Objects  Cases  Help  < CARSPECS >
-----1-----2-----3-----4-----5-----6-----7-----8-----
| - |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   |           Country      : _____
|   |           Car         : _____
|   |
|   |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   |   MODEL              BodyStack BODYTYPE    DEALER_COST  RETAIL_COST
|   |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   |   _____          _____          _____          _____
|   |   _____          _____          _____          _____
|   |   _____          _____          _____          _____
|   |
|   |           Select country and car, then update old cost data.
|   |
|   |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   |-----1-----2-----3-----4-----5-----6-----7-----8-----9-----12-----
|   | F1=Help 2=AddLn 3=Exit 4=NewFld 5=Move 6=DelObj 7=Up 8=Down 9=DelLn 12=EdtObj

```

Once you have completed the application, users will be able to select a country and car by using buttons that you will add in Step 5: Adding Buttons and Triggers.

Step 5: Adding Buttons and Triggers

So far you have created a Winform that displays data, that is, the Car and Country fields and the grid displaying BodyStack columns. However, you have not created a way to traverse the data or to update the data source. You cannot process the data.

In this step you add triggers and command buttons to provide a way to process the data. A trigger is a procedure that is invoked—“triggered”—by a specified event. The procedure can be a function or a system action. (Triggers are also known as event handlers, and functions are referred to as cases in the Winform Painter.) The event can be something the user does in a Winform, such as clicking a button. For example, a user clicking a Cancel button triggers a system action that closes the Winform.

In this section, you add buttons and triggers to the Winform and assign the triggers to the buttons. By clicking one of these buttons, the user triggers a function that pages through the data and updates the data source, or invokes a system action that closes the Winform or exits the application.

Adding the First Button and Trigger

The first step is to add a command button to the Winform. This button triggers a function that moves backward through the data, displaying values for the previous car brands.

Try it now:

1. Select the *Button* option from the Objects menu.
2. In response to the prompt at the bottom of the screen to define the top left corner of the button, position the cursor at row 20, column 5 (using the terminal emulator's coordinates) and press Enter.
3. In response to the next prompt, to define the button's bottom right corner, position the cursor at row 20, column 20, and press Enter.

The Button dialog box opens.

```
Button
Text:
Justification: ( ) Left
               ( * ) Center
               ( ) Right
Trigger: Cases F5
Shortcut key (PF1-PF24): PFKeys F6
[ ] Default Button [ ] Border
Object Name: Button1
Ok F4 Cancel F3
```

Text

You can type descriptive text to be displayed on the face of the button in the Text entry field. This text usually describes what the button does and includes the name of the button's shortcut key.

Try it now: Type the following text and include two spaces between the description and key name for reading ease: *Previous Car F1*.

Justification

The Justification radio button group determines where the text is aligned on the button.

Try it now: Accept the default justification so that the text will be centered.

Trigger, Functions

You specify a function to be performed or a built-in Winform system action to be called when the application user clicks the button. (Functions are referred to as cases in the Winform Painter.) To specify a *system action*, you press PF5 or click *Cases* to display a list of system actions and existing functions. After you select the desired system action, the Painter copies your choice to the Trigger field.

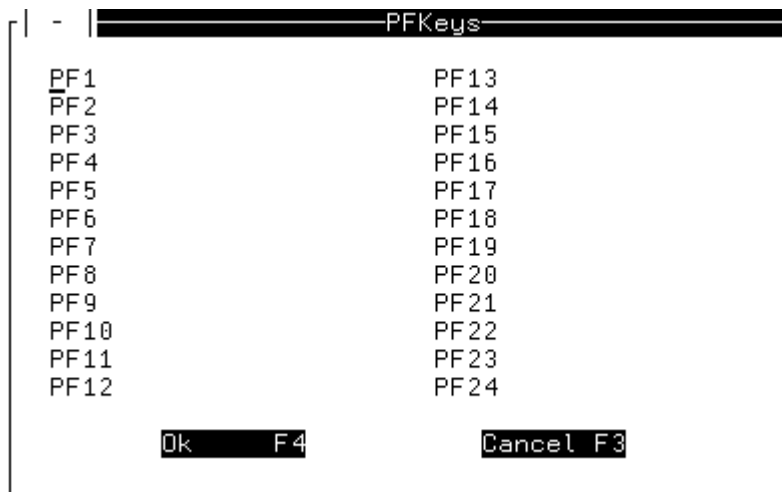
To specify an *existing function*, press PF5 or click *Cases* to display a list of system actions and existing functions. After you select the desired function, the Painter copies your choice to the Triggers field and assigns that function to the button.

To specify a *new function*, type the name of the function into the Triggers field.

Try it now: For the tutorial, you must specify a new function, so type *PrevCar* in the Trigger field.

Shortcut Key Field; PFKeys

The Shortcut entry field enables you to specify a function key that users can press to click the button, thereby saving keystrokes. If you wish to assign a shortcut key, first determine which keys are available by clicking *PFKeys*. This displays a list of the function keys (within this Winform) already assigned to buttons and to form-level triggers.



Try it now: Select *PF1* in the PF Keys dialog box and then click *OK* to confirm your choice.

The Painter copies the selected key name to the Shortcut key field and later (upon closing the Button dialog box) assigns PrevCar (your trigger).

You can change the values assigned to other function keys—for example, moving a default system action to a different key—by selecting the Actions option from the Forms menu.

Default Button

In some Winforms, one command button is the default. You can click the default button quickly by pressing Enter. On terminals that support color, the default button is the color of the terminal's unprotected high intensity field attribute. It is helpful to make the most frequently used button—for example, the OK button—the default.

Try it now: Leave the *Default* attribute unchecked, as you do not want this button to be the default.

Border

This attribute displays a border around the perimeter of the button. This is useful if your terminal emulator does not support highlighting.

Try it now: Accept the default value of no border, as you do not want these buttons to have borders.

Object Name

Try it now: Accept the default name.

When you are finished specifying in the Button dialog box, it should look similar to the following:

```

| - | Button
      Text:
      Previous Car F1
      Justification:  ( ) Left
                    ( * ) Center
                    ( ) Right
      Trigger: PrevCar      Cases F5
      Shortcut key (PF1-PF24): PF1      PFKeys F6
      [ ] Default Button  [ ] Border
      Object Name: Button1
      Ok F4      Cancel F3
  
```

Try it now:

1. Click *OK* to confirm your work and close the dialog box.

The Painter displays a message that the `PrevCar` function (which you had specified as the button's trigger) does not exist and asks if you want the Painter to create the function.

2. Click the *Create* button.

The Painter automatically generates an empty `PrevCar` function in the `Maintain` procedure.

This generated function (consisting of `CASE` and `ENDCASE` commands) serves as a temporary placeholder while you continue to develop the application.

The Painter generates code when you save your work. To demonstrate what the Painter produces, save your work by selecting the *Save* option from the `File` menu.

Try it now:

1. Place the cursor on the button you just created and press PF12 to edit the button. When the Button dialog box opens, press PF6 to display the PFKeys list.

Notice that PF1 now displays the button's trigger. (Use PF11 and PF10 to scroll the display to see the entire trigger).

2. Click *Cancel* twice to close the PFKeys and Button dialog boxes.

Adding Additional Buttons and Triggers

Now you will add three additional command buttons to the Winform. Select the *Button* option from the Objects menu for each one, and supply the following values:

Try it now: Start the second button in row 20, column 24 and end it in row 20, column 39.

1. Enter *Next Car F2* as the button's text.
2. Type *NextCar* in the Trigger field.
3. Select *PF2* for the shortcut key.
4. Accept the defaults for Justification, Border, Default, and Object Name.
5. Click *OK* to close the dialog box.
6. When the Painter notifies you that the *NextCar* function does not exist, click *Create* to generate an empty function.

Try it now: Start the third button in row 20, column 43, and end it in row 20, column 58.

1. Enter *Quit F3* as the button's text.
2. Supply a value for the Trigger field by clicking the *Cases* button and selecting the *exit* system action. Select *PF3* for the shortcut key. Accept the defaults for Justification, Border, Default, and Object Name. Click *OK* to close the dialog box.
3. You may see the following message: This key is already assigned to a system action. Do you want to override it? You can select *OK*.

Try it now: Start the fourth button in row 20, column 62, and end it in row 20, column 77.

1. Enter *Done F4* as the button's text.
2. Supply a value for Trigger by clicking *Cases* and selecting the *close* system action.
3. Select *PF4* for the shortcut key.
4. This will be the default button, so select the *Default* check box.
5. Accept the defaults for Justification, Border, and Object Name. Click *OK* to close the dialog box.

The Winform now displays four buttons:

The screenshot shows a window titled 'CAR SPECS' containing a 'Car Editing Form'. The form has a menu bar with 'File', 'Edit', 'Forms', 'Objects', 'Cases', and 'Help'. A 'ShowCars' button is visible in the menu. Below the menu are two input fields: 'Country : _____' and 'Car : _____'. A table is displayed with the following columns: MODEL, BodyStack, BODYTYPE, DEALER_COST, and RETAIL_COST. Below the table is the instruction: 'Select country and car, then update old cost data.' At the bottom of the form are four buttons: 'Previous Car F1', 'Next Car F2', 'Quit F3', and 'Done F4'. A status bar at the bottom of the window lists keyboard shortcuts: F1=Help, 2=AddLn, 3=Exit, 4=NewFld, 5=Move, 6=DelObj, 7=Up, 8=Down, 9=DelLn, 12=EdtObj.

Step 6: Coding Triggers and Other Functions

There are two ways you can supply the function logic needed for the Winform's triggers:

- Use TED (or, under CMS, any editor of your choice) to edit the Maintain procedure and add the appropriate logic.
- Use the Painter's Cases menu to add new functions and edit existing ones. This enables you to do all of your application development—form building and function coding—within a single environment (that is, within the Painter).

When you are finished, Maintain must have the following functions:

- An expanded Top function that retrieves Car and Country data and places the output into the CarStack stack. It also performs GetBody and updates the data source when the Winform is exited.
- A new GetBody function that retrieves data from the CarRec and Body segments and places it into the BodyStack stack.
- A PrevCar function that updates the Body segment using the data from the BodyStack stack. It then retrieves data from the CarRec and Body segments for the previous Car. This newly retrieved data is displayed.
- A NextCar function that updates the Body segment using the data from the BodyStack stack. It then retrieves data from the CarRec and Body segments for the next Car. This newly retrieved data is displayed.

Painter-generated Code

Before adding any new functions, it is important to understand the code the Winform Painter already has generated. As you are building your screen, the Winform Painter is adding code in your Maintain procedure. If you were to save and exit the Painter now, the following code would be generated for you in the Maintain procedure CARSPECS.

Note: Do *not* change anything on or following the `-* >> Generated Code Section` line. This section contains code and comments generated by the Winform Painter.

```
MAINTAIN FILE CAR
case PrevCar
endcase

case NextCar
endcase

-* >> Generated Code Section ...

.
.
.

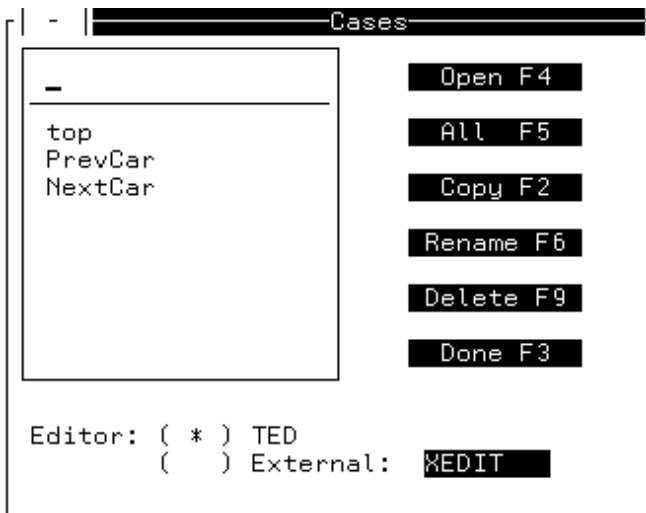
END
```

Top Function

Now you will add code to the Top function.

Try it now: Select *Cases* from the menu bar.

The Cases dialog box opens:



Under CMS, you can use TED (the default) or a different editor. You specify which alternative editor is available using the Preferences option of the File menu. Under MVS, only TED is available. In this tutorial, use TED.

Try it now: To edit the Top function, select *top* in the combo box. After *top* appears in the entry field, click the *Open* button. The Painter invokes TED and displays the following editing screen:

```

TEMP00  FOCEXEC  A1                               SIZE=0    LINE=0
* * * TOP OF FILE * * *
* * * END OF FILE * * *

=====> _
T E D

```

The file name in the upper left corner (TEMP00) is a default name for the temporary file that TED uses to edit functions. It doesn't affect your editing session.

You can use TED's standard editing facilities.

Do not enter the numbers (annotations) at the beginning of each line.

Try it now: Add the following code to the Top function:

1. FOR ALL NEXT Country Car INTO CarStack;
2. PERFORM GetBody;
3. WINFORM SHOW ShowCars;
4. FOR ALL UPDATE Dealer_Cost Retail_Cost FROM BodyStack;

1. Retrieves all the Country and Car combinations into CarStack. The NEXT command specifies which segments to retrieve, and the INTO phrase specifies into which stack to place the retrieved rows. The FOR ALL prefix specifies that all of the rows in the data source must be retrieved.

- 2. Performs** the GetBody function. GetBody retrieves these same keys, as well as Model, BodyType, and all the non-key fields in the Body segment into BodyStack. It retrieves only those records (also known as path instances or relational rows) whose Car and Country fields match the values in the current row of CarStack. The grid that you created displays selected columns from BodyStack.
- 3. Displays** the ShowCars Winform. From this Winform, the application user can browse and edit data using function key and button triggers.
- 4. Writes** the application user's last round of edits to the data source after the user leaves the Winform.

For more information on any of the Maintain commands or system variables included here, see Chapter 6, *Language Rules Reference*; Chapter 7, *Command Reference*; Chapter 8, *Expressions Reference*; and the *Using Functions* manual.

Try it now: To save your work and return to the Cases dialog box, enter *FILE* at the TED command line.

PrevCar Function

Try it now: Select the *PrevCar* function from the dialog box and click *Open*.

Maintain displays the following TED editing screen:

```
TEMP00  FOCEXEC  A1                               SIZE=3    LINE=0

* * * TOP OF FILE * * *

case PrevCar
endcase
* * * END OF FILE * * *

====> _                                           T E D
```


Try it now: Add the following function that enables users to scroll backward through the data and write their last changes to the data source each time they scroll. (Changing the generated CASE and ENDCASE keywords to uppercase is optional and is done for consistency. It does not affect execution.)

```

CASE PrevCar
1.  FOR ALL UPDATE Dealer_Cost Retail_Cost FROM BodyStack;
2.  IF CarStack.FocIndex GT 1
2.  THEN COMPUTE CarStack.FocIndex=CarStack.FocIndex-1;
2.  ELSE COMPUTE CarStack.FocIndex = CarStack.FocCount;
3.  PERFORM GetBody;
ENDCASE

```

1. **Writes** the application user's last Dealer_Cost and Retail_Cost changes from BodyStack to the data source. BodyStack contains values for all the records defined by the Country and Car values in the current CarStack row.
2. **Displays** the previous values of Car and Country by changing the current row of CarStack to the previous row.

Winforms display data from stacks based on the value of the stack variable FocIndex. The command reads: if the row the user is currently viewing (indicated by CarStack.FocIndex) is greater than 1, then subtract one from FocIndex; otherwise set FocIndex equal to the number of rows in the stack.

FocCount is another stack variable that automatically keeps track of the number of rows in the stack. This means when the PrevCar function is executed, the prior row is displayed unless the user is positioned on the first row in the stack. In that case, the last row is displayed. In this way, the user can cycle through the data.

3. **Performs** the GetBody function. GetBody retrieves fields from the first four segments of the Car data source into BodyStack, where they can be displayed in the grid you created. Only records whose Car and Country fields match the key values in the current row of CarStack are retrieved.
4. To save your work and return to the Cases dialog box, enter *FILE* at the TED command line.

NextCar Function

Try it now: Select the *NextCar* function from the Cases dialog box and click *Open*. Enter the following function:

```
CASE NextCar
  FOR ALL UPDATE Dealer_Cost Retail_Cost FROM BodyStack;
  IF CarStack.FocIndex LT CarStack.FocCount
    THEN COMPUTE CarStack.FocIndex = CarStack.FocIndex + 1;
  ELSE COMPUTE CarStack.FocIndex = 1;
  PERFORM GetBody;
ENDCASE
```

This function is almost identical to the *PrevCar* function. The only differences are in the COMPUTE and IF commands. Rather than subtracting one from the value of *CarStack.FocIndex*, it adds one. This is because you want to display the next Country Car combination. The value of *CarStack.FocIndex* determines which row is displayed in the Winform.

Refer to the comments for the *PrevCar* function for additional information.

Try it now: To save your work and return to the Cases dialog box, enter *FILE* at the TED command line.

GetBody Function

Now you will enter a new function named *GetBody*. It is not invoked by any of the Winform's triggers, so the Painter has not generated any code for it. Instead, this function is called by the other functions (*Top*, *NextCar*, and *PrevCar*) when they need to retrieve new values into *BodyStack* that match the key values in the current row of *CarStack*.

Try it now: Type *GetBody* in the Cases entry field and click *Open*. Enter the following code:

```
CASE GetBody
1.  STACK CLEAR BodyStack;
2.  REPOSITION Country;
3.  FOR ALL NEXT Country Car Model BodyType INTO BodyStack
3.  WHERE CarStack().Country EQ Car.Country
3.  AND CarStack().Car EQ Car.Car;
ENDCASE
```

1. **Clears** the existing values from *BodyStack*.
2. **Repositions** the application to the beginning of the data source, ensuring that the next set-based data source retrieval selects records from the entire data source.

- 3. Retrieves** fields from the first four segments of the Car data source into BodyStack. Only records whose Car and Country fields match the key values in the current row of CarStack are retrieved.

The FOR ALL prefix means that all rows in the data source are examined to see if they meet the selection criteria. The rows to be retrieved are specified in the WHERE phrase.

The components of the WHERE phrase are:

`CarStack().Country`

Indicates the value of the Country column—from the CarStack stack—that is currently displayed. This is a qualified field name (actually, in this case, a qualified column name).

CarStack is the name of a stack that contains the Country column.

CarStack() indicates the row of CarStack that is currently displayed and is shorthand for CarStack(CarStack.FocIndex). FocIndex is a system variable whose value is always the current stack row.

`CarStack().Car`

Indicates the value of the Car column—from the CarStack stack—that is currently displayed.

`EQ`

Only the data source rows matching the data in the stack are retrieved.

`Car.Car`

Refers to the Car field in the Car data source. This is a qualified field name.

The first Car is the name of the data source, and the second is the name of the field.

`Car.Country`

Refers to the Country field in the Car data source.

For more information on any of the Maintain commands or system variables included here, see Chapter 6, *Language Rules Reference*; Chapter 7, *Command Reference*; Chapter 8, *Expressions Reference*; and the *Using Functions* manual.

Try it now: To save your work and return to the Cases dialog box, enter *FILE* at the TED command line. Then click *Done* to exit Cases. Press PF3 to exit the Painter. When prompted about saving your work, click *Yes*.

If you wish to view a complete Maintain procedure built by the Painter, including the functions you entered, Painter-generated code, and Painter-generated comments, you can issue the command

`TED CARSPECS`

at the FOCUS command line to see the entire Maintain procedure.

Step 7: Running the Maintain Request

Try it now: At the FOCUS prompt, enter the following command:

```
MNTCON EX CARSPECS
```

or

```
EX CARSPECS
```

The CarSpecs application displays the following Winform:

```
Car Editing Form
Country      : ENGLAND
Car          : JAGUAR

MODEL        BODYTYPE    DEALER_COST
1 V12XKE AUTO  CONVERTIBLE  7,427
2 XJ12L AUTO   SEDAN        11,194

Select country and car, then update old cost data

Previous Car F1  Next Car F2  Quit F3  Done F4
```

The application's function keys perform the following actions:

PF7, PF8, PF10, and PF11 are default system actions preassigned to those keys.

PFKey	Action
PF1	Data for the previous car displays. The display is cycled through the entire stack. This means that if you are positioned at the first row, your position is changed to the last row.
PF2	Data for the next car displays. As with PF1, the display is cycled so if you are positioned at the last row, your position is changed to the first row.
PF3	Enables the user to exit the procedure. All changes since the last data source update are lost. (The data source is updated when you press PF1, PF2, or PF4.)

PF Key	Action
PF4	Exits the procedure, and the data source is updated with any changes made.
PF7	When the cursor is on the grid, the rows in the grid are scrolled backward.
PF8	When the cursor is on the grid, the rows in the grid are scrolled forward.
PF10	When the cursor is on the grid, the columns in the grid are scrolled left.
PF11	When the cursor is on the grid, the columns in the grid are scrolled right.

Congratulations! You have painted an event-driven Maintain application.

Step 7: Running the Maintain Request

CHAPTER 5

Using the Winform Painter

Topics:

- Using the Painter
- Files Used by the Winform Painter
- Saving and Exiting Your Work
- Using the Design Screen
- File Menu
- Edit Menu
- Forms Menu
- Objects Menu
- Cases Menu
- Help Menu
- Using Triggers, Button Short Cuts, and System Actions

The Winform Painter is an application development environment that allows you to design and create event-driven applications. While in the Painter, you can design the visual layout of Winforms and define and code triggers.

Using the Painter

This section provides an overview of how to communicate with the Winform Painter, how to begin and end a Painter session, how to save your work, and which files are produced by the Painter.

To select any item in the Winform Painter, or in a Winform at run time—for example, to select a menu option, an item in a list box, a radio button, or to click a button—position the cursor on that item and then press the Enter key. (Pressing a function key has the same effect as pressing Enter, but will also perform whatever action is associated with that function key.)

Because the Winform Painter runs in a mainframe environment, the host computer does not become aware of what you (or an end user) do on the screen until you press the Enter key or a function key.

Using Dialog Boxes

A dialog box is used to request information about a task you are performing, or to supply needed information.

For example, a dialog box is displayed if you choose *Field* from the Objects menu. In the Create Field dialog box, you specify the field name, its input (or source), its output (or destination), and other information about displaying the field.

The screenshot shows a dialog box titled "Create Field". On the left, there is a list of fields: CUSTID, LASTNAME, FIRSTNAME, EXPDATE, PHONE, STREET, CITY, STATE, ZIP, and TRANSDATE. The "Field" field is selected. To the right of the list, there are sections for "Source:" and "Destination:". Under "Source:", there are radio buttons for "Default", "Current Area", and "Stack" (which is selected). Below these is a "Source Stack" dropdown menu with "stkcust" selected. Under "Destination:", there is a "Destination Stack" dropdown menu with "stkcust" selected. To the right of these sections, there is a "Default value:" field, a "Length =" field, and checkboxes for "Prompt:", "Uppercase", and "Protected". At the bottom, there is an "Object Name:" field with "Field1" entered. On the right side of the dialog, there are several buttons: "Ok F4", "Accepts F6", "Triggers F12", "Color F5", and "Cancel F3".

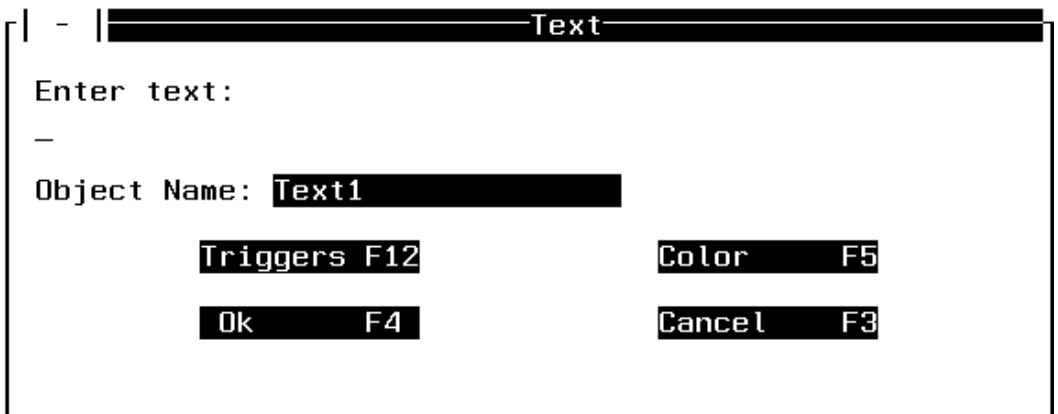
Most dialog boxes contain options you can select. After you specify options, you can choose a command button to carry out a command.

Often you need to move within a dialog box to select one or more options. When you enter the dialog box, the cursor is on the first option. To move to another option, you can press the Tab key to move forward through the options or you can press the Tab and the Shift key to move backward through the options. Inside a dialog box, PF3 is always Cancel or No and PF4 is always OK or Yes. When a dialog box offering several options has a default option, Enter always selects the default.

The kinds of controls that you may see in a dialog box are described in the following sections.

Using Entry Fields

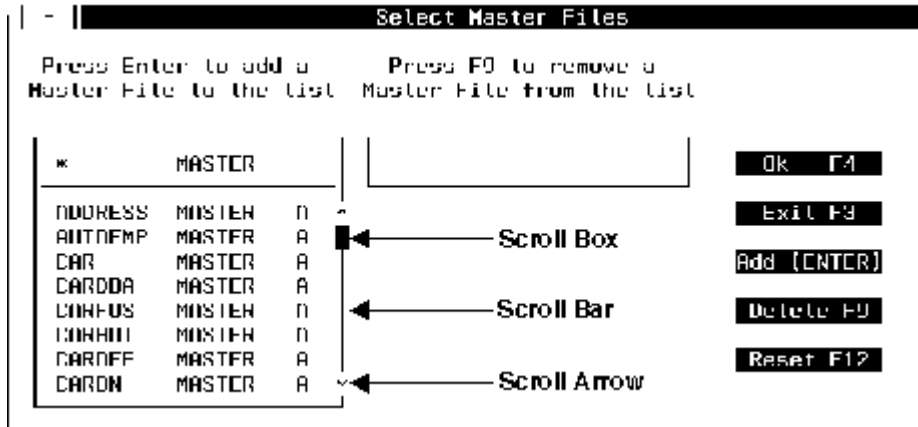
You type information into an entry field.



When you move into an empty entry field, the cursor appears on the left side of the box. Whatever is typed into the entry field is started where the cursor is. You can enter more text than will fit in the box by scrolling the text. This is done by positioning the cursor at the end of the editable area and pressing PF11. PF10 scrolls left and PF11 scrolls right.

Using List Boxes

A list box displays a list of choices. If there are more choices than can fit in the box, scroll bars are provided so you can move through the list. The selected item is displayed in a different color than the rest of the list on terminals that support color. The following shows a list box (with scroll bar annotations) displaying a list of all of the Master Files available.



In list boxes and anywhere else scroll bars might appear on the screen, scroll bars can be used as follows:

To scroll ...	Move the cursor to ...
One line up or down.	The up or down arrow on the vertical scroll bar and press the Enter key.
One screen up or down.	Just above or below the arrow on the vertical scroll bar and press the Enter key.
One position left or right.	The right or left arrow on the horizontal scroll bar and press the Enter key.
One screen left or right.	Just to the right or left of the arrow on the horizontal scroll bar and press the Enter key.

When you scroll through a list, the scroll box moves up, down, left or right to indicate where you are in the list.

To select an item from a list box, move through the list as specified above. When the item you want is displayed, move the cursor to that entry and press the Enter key.

Using Check Boxes

A check box indicates that you can select or clear (that is, deselect) the indicated option. You can select an option by typing any character in the middle of the check box; after you press the Enter key or a function key, the check box is shown with an X in the middle. You can clear an option by removing the X from the check box.

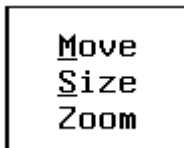
Check boxes are displayed using square brackets or angle brackets, depending on your hardware. If strange characters appear instead of brackets, select the *Terminal* option from Preferences in the File menu to get something more appropriate to display. For more details see *Preferences* on page 5-25.

[X] Uppercase

[] Protected

Using the Control Box

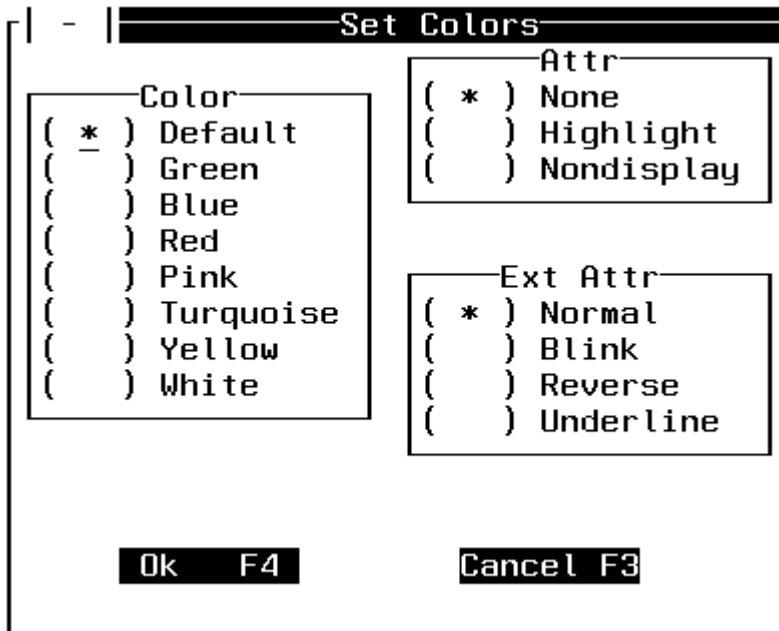
The Control box is used to alter the size of, or move a Winform. The Control box is in the upper left-hand part of the Winform. It is a dash framed by two parallel lines. If you position the cursor between the two parallel lines and press the Enter key, the Control box menu is displayed:



At run time the control box allows you to move or close the Winform but not size or zoom it. For details about Move, Size and Zoom, see *Size* on page 5-42, *Zoom* on page 5-42 and *Move* on page 5-43.

Using Radio Buttons

Radio buttons represent mutually exclusive options. You can select only one radio button at a time. A radio button is indicated by parentheses, and the selected button displays an asterisk.



To select a radio button, move the cursor to the button—to the middle position between the button's parentheses—and type any character. When you then press the Enter key or a function key, that button is selected, and it displays an asterisk.

If more than one option is selected, only the last option is considered selected and all other options are left blank. If the keyboard locks after you type a key, it probably means that the cursor was not in the middle of the radio button. If this happens, press the Reset key and try again.

Using Command Buttons

You click a command button to initiate an immediate action, such as carrying out or canceling a command. The *OK* and *Cancel* buttons are common command buttons. They usually are located along the bottom or on the right side of the dialog box. To click a command button, move the cursor to the button and then press the Enter key. Or, if the button has a shortcut key, you can click the button even faster simply by pressing its key. (A shortcut key is a function key assigned to a button. When a button has a shortcut key, the key name is usually displayed next to the button name.) In the sample screen, the *Cancel* button's shortcut key is F3: you can click the button either by pressing F3, or by moving the cursor to the *Cancel* button and pressing Enter.

In some dialog boxes, one command button is the default. In addition to the standard ways to click a button described previously, you can always click the default button by pressing Enter. On terminals that support color, the default button is the color of the terminal's unprotected high intensity field attribute.

The following shows the command buttons at the bottom of the Listbox dialog box:

Triggers F12

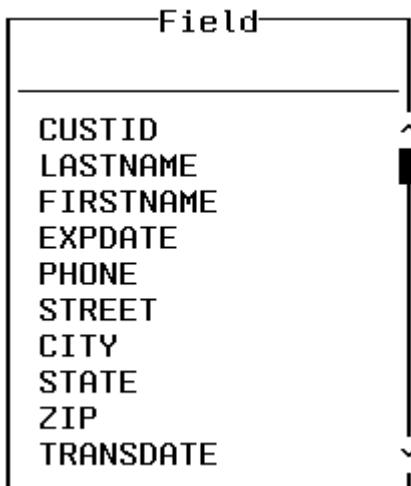
Ok F4

Cancel F3

Using Combo Boxes

When an entry field appears together with a list box, the combination is known as a combo box. You can place a value into the entry field by selecting it from the list box or by typing it directly into the entry field.

The following shows the combo box in the Create Field dialog box:

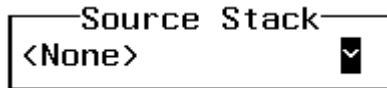


Note that combo boxes that you create in a Winform are similar to drop-down list boxes.

Using Drop Boxes

A drop box is an entry field with a down-arrow (v) button. When you click the button, the Painter displays a list box directly adjacent to the display field (as if the list had dropped down from the field). When you select a value from the list box, the Painter copies it to the display field and removes the list box.

The following shows the initial display field in the Create Field dialog box:



After clicking the button, the drop box is displayed:



Supporting Colors

The ability to display colors depends upon your terminal, or if you are using a PC, your terminal emulation software, and on describing it correctly to Maintain. This is discussed in *Preferences* on page 5-25.

Supporting Borders

Winforms can have borders around them. The physical look of these borders is determined by your terminal or your terminal emulation software. If your terminal is not capable of generating solid lines, your work may not match some of the examples in this document. The only difference is in appearance, not functionality. You can adjust the settings for borders. This is discussed in *Preferences* on page 5-25.

Files Used by the Winform Painter

The Winform Painter uses the following files:

- **A Master File** for each data source accessed by the Maintain procedure. The Master Files required by a Maintain procedure must exist before you begin painting. See the *Describing Data* manual for more information about creating and using Master Files.
- **A WINFORMS file** that contains all of the Maintain procedure's Winforms. (Winforms are also known as forms.) The Painter creates this file the first time that you begin painting the procedure.
- **A Maintain procedure file** that contains the procedure's Maintain request. If the Maintain procedure file does not exist when the developer begins painting the procedure, the Painter creates it. This file has the extension of either MAINTAIN or FOCEXEC.

Each procedure's Maintain procedure and WINFORMS files have the same file name (under CMS) or member name (under MVS). This is the name of the procedure. For example, the Billing procedure might be made up of the files BILLING FOCEXEC and BILLING WINFORMS.

How to Access the Painter

To access the Winform Painter, enter MPAINT at the FOCUS prompt. If you wish, you can specify a particular Maintain procedure to paint:

```
MPAINT [procedure_name]
```

If:

- You do not specify a name, the Painter displays the Open Maintain dialog box that enables you to choose an existing procedure or create a new one. For information about using this dialog box, see *Open* on page 5-20.
- The named procedure exists (that is, both its FOCEXEC/MAINTAIN and WINFORMS files exist), the Painter displays the procedure's first Winform in the design screen. You can edit it or create additional Winforms for the procedure.
- The named procedure does not exist, or only one of its two files exists, the Painter creates the procedure. If one file exists, the Painter gives you a choice of retaining it (if it's a FOCEXEC or MAINTAIN), replacing it, or canceling the operation. The Painter then displays the following dialog boxes:

Select Master Files, which enables you to select the Master Files to be associated with the procedure. For information about using this dialog box, see *Select Master* on page 5-24. (This is not displayed if you choose to retain an existing FOCEXEC/MAINTAIN file.)

Winform Properties, which enables you to specify the properties of the first Winform. For information about using this dialog box, see *Properties* on page 5-37.

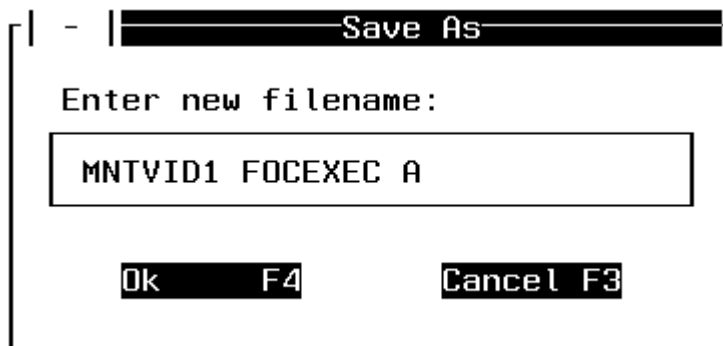
If you wish, later you can revise any information that you enter into the initial dialog boxes using the design screen's pull-down menus.

Saving and Exiting Your Work

It is a good idea to save your work from time to time. This way, if the machine you are working on is no longer available, you can pick up later on from your last save point.

Saving the Winform

Move the cursor to the menu bar by pressing the Home key or PF10. Press the Enter key so you can see the pull-down menu. Select either *Save* or *Save As* from the menu by moving the cursor to the appropriate item and pressing the Enter key. If you select *Save*, your keyboard will lock while the file is being saved. When it is unlocked, you can continue painting the screen if you want. There is also a shortcut key (PF22) for the Save option. This means that pressing PF22 saves your application. If you select *Save As*, the Save As dialog box is displayed.



The *Enter new filename* entry field will be filled in with the current Maintain procedure name but you can change it to any name you want. When you have filled in the name, press the Enter key and the file will be saved.

Procedure How to Exit the Winform Painter

There are two ways you can exit from the Winform Painter:

1. Press the Home key or PF10 to move the cursor to the menu bar, press the Enter key to see the pull-down menu, and then move the cursor to the *Exit* item.
2. Pressing PF3 exits you from whatever you are doing. This means that if a dialog box or a pull-down menu is being displayed, the dialog box disappears. If, on the other hand, you are not in the middle of doing something, pressing PF3 exits you from the Painter. If you have made any changes since the last time you saved the file, the Painter asks if you wish to do so before closing it. Move the cursor to the appropriate button and press the Enter key.



Using the Design Screen

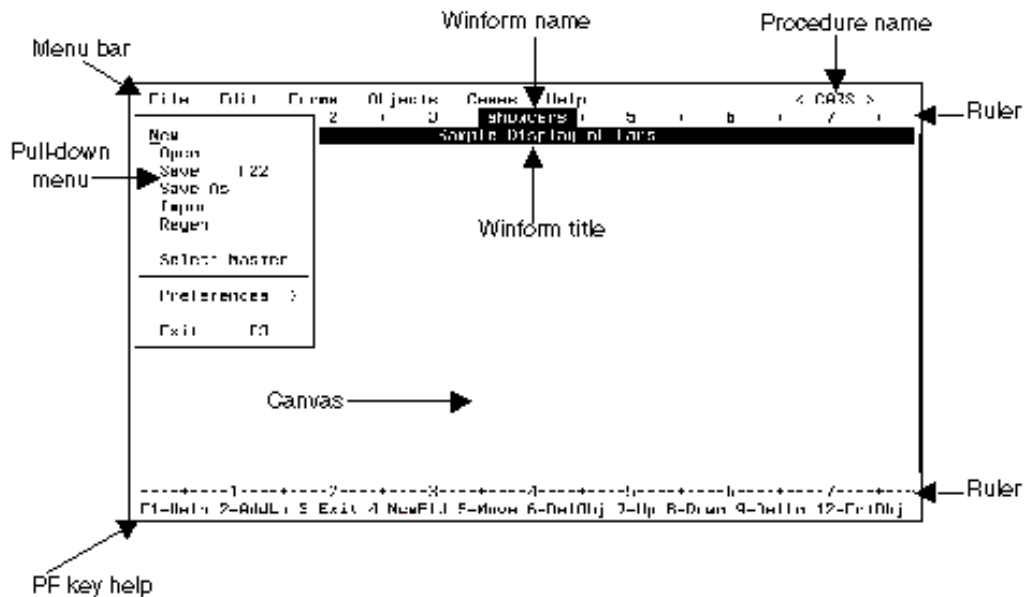
To edit a control on the design screen using a menu option:

1. Select the desired menu option.
2. Position the cursor on the desired control. (Controls, such as buttons and list boxes, are referred to in the Winform Painter as objects.)
3. Press the Enter key.

To edit a control on the design screen using a function key:

1. Position the cursor on the desired control.
2. Press the desired function key.

The following is a description of different elements that may be displayed on the screen when you are painting a Winform.



Menu Bar

Most of the Winform Painter's features are available from its menu bar. To select a menu option:

1. **Move to the menu bar.** Press the Home key or PF10 to position the cursor immediately before the first menu item.

Alternatively, you can use the arrow keys to move the cursor to the menu bar.

2. **Select a menu.** You can navigate through the menu bar using the:
 - Tab key to cycle through the menus and, if one of the menus is open, through that menu's options. Press Tab + Shift to move in the reverse direction.
 - PF10 and PF11 keys to move backward and forward, respectively, through the menus. If the cursor is on the menu bar, it moves to the previous or next menu; if the cursor is on an open menu, it moves to the previous or next menu and opens that menu.
 - Arrow keys.

Once the cursor is on the desired menu, press the Enter key. Most menus open into a pull-down menu from which you can select an option. (The Cases and Help menus each display a dialog box; they do not open into a menu of options.)

3. **Select a menu option.** Position the cursor on the desired option in the pull-down menu and press the Enter key.

If you wish to close a menu without selecting an option, press *PF3* while the cursor is positioned anywhere in the menu system. (Note that if the cursor is on the Winform Painter canvas, PF3 closes the Painter.)

Some menu options are shown with a function key (for example, F3) to the right of the option. This indicates that the function key is a shortcut key that you can press to select the menu option; when you press the function key, the cursor can be anywhere on the Winform Painter canvas. Pressing a shortcut key is a quick alternative to navigating through the menu bar and menu options.

Procedure Name

This is the name of the MAINTAIN file from which the Winform is displayed. The name is supplied by the developer from either the:

- New Maintain dialog box, or the
- Open Maintain dialog box.

Winform Name

Each Maintain procedure can display many different Winforms. All of the Winforms are stored in one file. This means that there needs to be a way to identify which Winform should be displayed, modified, or created. The name displayed here is the one supplied in the Winform Name entry field of the Winform Properties dialog box.

Winform Title

The Winform title is for the benefit of the application user, and is optional. The title is centered in the Winform's top border; if a Winform has no border, it does not display the title.

Ruler

Maintain's Winform Painter provides a ruler both at the top and the bottom of the screen in order to help the developer line up fields more easily.

Canvas

This is the area on which the developer paints the Winform.

PF Key Help

The line below the bottom ruler is reserved for a list of common function keys. If the Painter needs to instruct the developer to do something, the list is not displayed, and a message is displayed.

Reference **Function Key Reference**

The function keys are very useful in helping you to design and edit Winforms inside the Painter. When you are inside the Painter, the settings for some of the function keys appear at the bottom of your screen, as shown below:

F1=Help 2=AddLn 3=Exit 4=NewFld 5=Move 6=DelObj 7=Up 8=Down 9=DelLn
12=EdtObj

The following is a brief description of the function keys in the Painter. These function keys may operate differently in an individual pull-down menu.

Function Key	Usage
F1=Help	Offers help while inside the Painter.
F2=AddLn	Adds a line to a Winform.
F3=Exit	Exits the current operation.
F4=NewFld	Adds a new field to a Winform.
F5=Move	Moves a control within a Winform. (The Winform Painter refers to controls as objects.)
F6=DelObj	Deletes a control from a Winform.
F7=Up	Pages up in a Winform.
F8=Down	Pages down in a Winform.
F9=DelLn	Deletes a line from a Winform.
F10=Go To Main Menu	Goes to menu bar.
F11=Switch To Form	Allows you to switch into another Winform.
F12=EditObj	Edits a control within the Winform.
F13=Text	Adds a text control to the Winform.
F14=Grid	Adds a grid to the Winform.
F15=Browser	Adds a browser to the Winform.
F16=Frame	Adds a frame to the Winform.
F17=Button	Adds a button to the Winform.
F18=Check box	Adds a check box to the Winform.

Function Key	Usage
F21=Triggers	Adds a trigger to the Winform.
F22=Save	Saves current work to the WINFORMS file.
F24=Resize	Allows you to change the size of a control.
Home=Go To Main Menu	Goes to menu bar.
Tab=Move Cursor	Moves cursor across the screen, to the next control, left to right, and then down one line.
Shift + Tab=Move Cursor Backwards	Moves cursor across the screen, to the next control, right to left, and then up one line.

File Menu

The File menu enables you to create a new Winform file or to open a different Winform file. It also enables you to perform other actions within the Winform files. Selecting the File menu yields:

New	
Open	
Save	F22
Save As	
Import	
Regen	
<hr/>	
Select Master	
<hr/>	
Preferences	>
<hr/>	
Exit	F3

The File menu offers the following options:

- **New** enables you to create a new Winform file from an existing Winform.
- **Open** enables you to open an existing Winform file.
- **Save** enables you to save the work you have done in your current Winform file.
- **Save As** enables you to save the work you have done in your current Winform file under a different name.
- **Import** enables you to import another Winform into your Winform file. Import is very useful when you need a Winform in your Winform file that is similar to an already existing Winform.
- **Regen** enables you to make your Winform file compatible with a FOCUS upgrade.
- **Select Master** enables you to select all of the data sources you need in a Winform file.
- **Preferences** enables you to customize your Painter environment.
- **Exit** enables you to exit the Painter and return to the FOCUS prompt.

New

To close the Maintain procedure on which you are currently working, and create a new Maintain procedure, select *New*:

1. If you have not saved your work on the current procedure, the Painter asks if you wish to do so before closing it.
2. You are prompted for a file name for the new procedure's FOCEXEC/MAINTAIN and WINFORMS files. Enter the name and click *OK*.

If the FOCEXEC/MAINTAIN file or WINFORMS file already exists, you have a choice of retaining it (if it's a FOCEXEC/MAINTAIN), replacing it, or canceling the operation. If you choose to retain a FOCEXEC/MAINTAIN file, skip step 3.
3. The Painter displays the Select Master Files dialog box, as described in *Select Master* on page 5-24. Select the Master Files associated with this Maintain procedure.
4. The Painter displays the Winform Properties dialog box, as described in *Properties* on page 5-37. Specify the properties of the first Winform.
5. The Painter displays the Winform design screen: you can begin working on the new Maintain procedure.

Open

To close the Maintain procedure on which you are currently working, and open another one, select *Open*. The Painter displays the following dialog box:

Files: * FOCEXEC		
CAR	FOCEXEC	A
CARSPECS	FOCEXEC	A
NEW	FOCEXEC	A
MNTCALC	FOCEXEC	F
MNTCAR1	FOCEXEC	F
MNTCAR2	FOCEXEC	F
MNTCAR3	FOCEXEC	F
MNTCAR4	FOCEXEC	F

This is the same dialog box used by the Painter when you begin a session, as described in *How to Access the Painter* on page 5-11. You can do one of the following:

- **Enter a procedure name.** Type in the name of the Maintain procedure (adding the word FOCEXEC or MAINTAIN is optional) in the FOCEXEC Name entry field and click *OK*.

If you enter the name of a procedure that does not exist (that is, if the procedure's FOCEXEC/MAINTAIN file and/or WINFORMS file does not exist), the Painter creates a new procedure. If only one file of the pair does not exist, the Painter gives you the option of retaining the existing file (if it's a FOCEXEC/MAINTAIN), replacing the existing file when the Painter creates the new procedure, or canceling the operation. The Painter then displays the Select Master Files dialog box (unless you chose to retain an existing FOCEXEC/MAINTAIN file), which is described in *Select Master* on page 5-24; and the Winform Properties dialog box described in *Properties* on page 5-37.

You can also create a new procedure by clicking *New*. This is identical to selecting the File menu's *New* option.

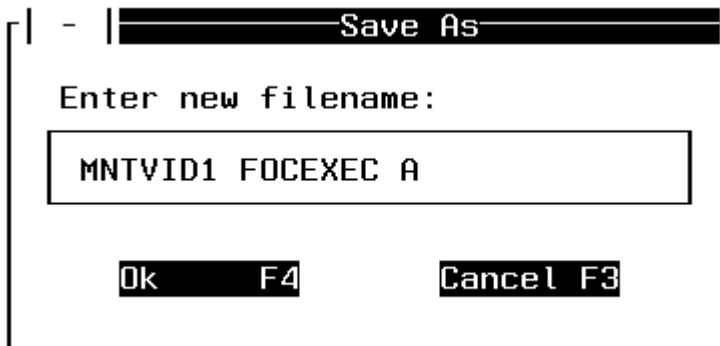
- **Select a procedure name.** Select a procedure from the list of files. If you wish to reduce the length of the list, you can use search criteria. For example, if you want to see all files that start with the letter M, you could enter M*. When you click *OK*, the list changes.
- **Exit.** If you click *Cancel* you cancel the action and close the dialog box.

Save

To save the work you have done in your current Winform file, select *Save* or press PF22.

Save As

To save the Winform file you are working with under a different name, select *Save As*. The *Save As* dialog box is displayed:



Enter the new name for the Winform file and press PF4 to save it. You may press PF3 to cancel the action.

Import

To bring an existing Winform into your Winform file, select *Import*. The Import dialog box is displayed:

Enter FOCEXEC name:

* FOCEXEC

Files: * FOCEXEC

CAR	FOCEXEC	A
CARSPECS	FOCEXEC	A
NEW	FOCEXEC	A
MNTCALC	FOCEXEC	F
MNTCAR1	FOCEXEC	F
MNTCAR2	FOCEXEC	F
MNTCAR3	FOCEXEC	F
MNTCAR4	FOCEXEC	F

Ok F4

Cancel F3

Type or specify a Maintain procedure name or move the cursor to the Maintain procedure you wish to import, then press PF4.

If the Maintain procedure name you specify has a different corresponding Master File, a dialog box similar to the following is displayed:

Warning

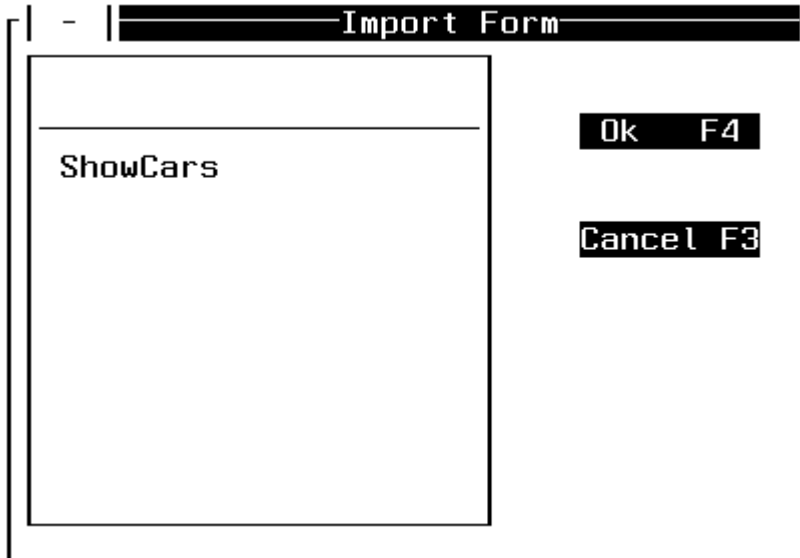
Different Master Files:

NEW: VIDEOTRK CAR: CAR

Ignore Cancel

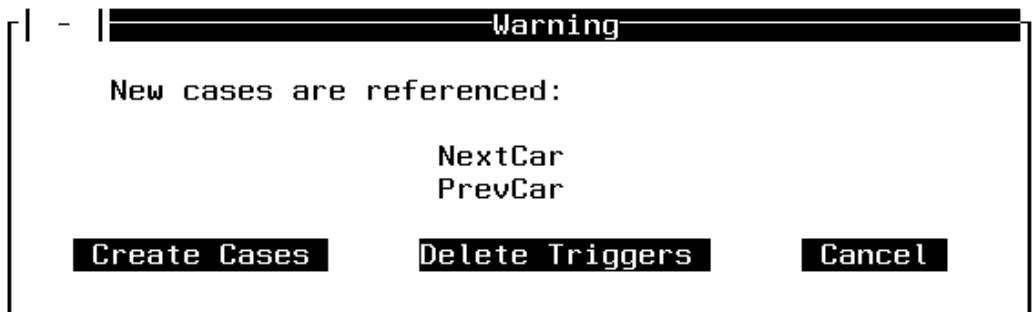
You may ignore the warning and proceed, or cancel the request.

The Painter now displays a list of the Winforms that you can import:



Select a Winform from the combo box and then click *OK*.

If the Winform you attempt to import references Maintain functions that do not currently exist in your procedure, a dialog box similar to the following is displayed. (Maintain functions—often simply called functions—are referred to in the Winform Painter as cases.)



In response to the warning you may either choose:

- **Create Cases** to generate the CASE and ENDCASE commands in your Maintain procedure, and to bring the triggers into your Winform. After selecting *Create Cases* you are returned to the Winform.
- **Delete Triggers** to delete the triggers that are not system actions. After selecting *Delete Triggers* you are returned to the Winform.
- **Cancel** to cancel the action. After selecting *Cancel* you are returned to the Winform.

Regen

Select *Regen* whenever a new error occurs after a new release of FOCUS is installed at your site. You should also select *Regen* whenever changes are made to your data source. The error may be occurring because of the new install of FOCUS. If the error still occurs after selecting *Regen*, consult your local Information Builders representative.

Select Master

Selecting *Select Master* displays the Select Master Files dialog box, enabling you to select all of the data sources you wish to access in a Winform file. This is the same dialog box used by the Painter when you begin a session, as described in *How to Access the Painter* on page 5-11.

* _	MASTER	A
ADDRESS	MASTER	A
AUTOEMP	MASTER	A
CAR	MASTER	A
CARDBA	MASTER	A
CARFUS	MASTER	A
CARHOT	MASTER	A
CAROFF	MASTER	A
CARON	MASTER	A

When the dialog box is first displayed, your cursor is positioned so you can type in the name of the file. You can either type in the name (and in this case the word MASTER is optional) or you can select from the list box which is just below the cursor. Once you have selected or entered a data source, you must press the Enter key to add the data source to the list. When you do this, the data source name is added to the box to the right of the list of available Master Files. You may select up to 16 data sources.

Your selections determine which fields are displayed in the Field and Grid dialog boxes.

If you accidentally add a data source you did not want to add, you can remove it from the list by moving the cursor to the data source name you want removed from the list of the selected data sources and pressing PF9. If you have made a lot of mistakes, you might want to press PF12 (or select the *Reset* button) which clears all changes made to the dialog box since it was displayed.

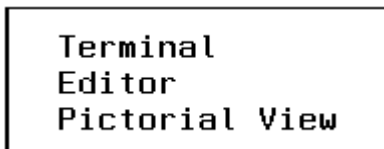
If this dialog box was displayed while beginning a Painter session for a new procedure, and the procedure will not access a data source, you do not need to select one. When this dialog box is displayed, press PF4 or the *OK* button.

When you have finished adding all of the data sources you want to access, click *OK*.

Preferences

Selecting *Preferences* enables you to customize your Painter environment. For all Preference options, the options selected stay in effect from session to session until you change them.

The following menu is displayed when *Preferences* is chosen:

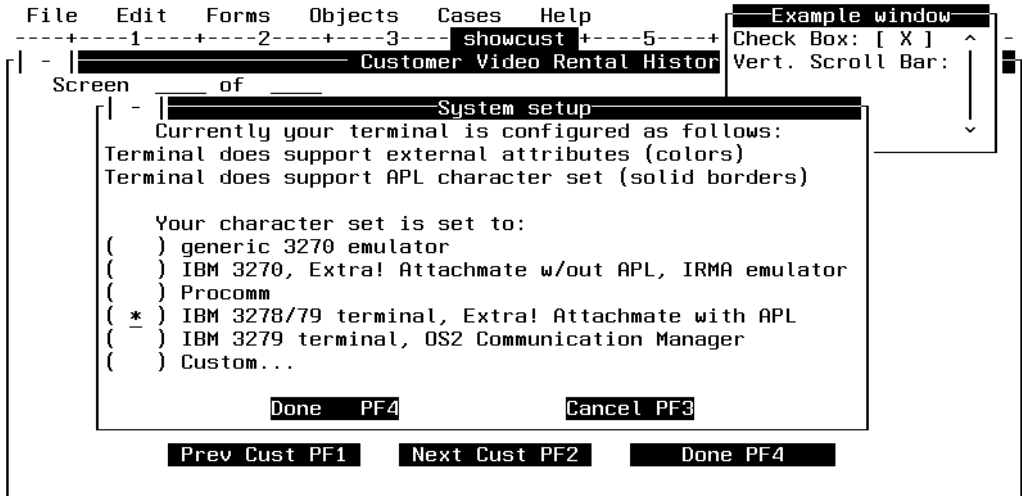


Preferences offers the following options:

- **Terminal** configures the Winform facility for the terminal or terminal emulator that you are using. If you are having problems displaying Winform elements such as check boxes, scroll bars, borders, or colors, you can set the Winform facility to a different terminal or emulator. (Selecting this option is equivalent to issuing the EX MSETUP command at the FOCUS command prompt.)
- **Editor** specifies the editor that the Painter invokes when you create or edit Maintain functions. You can use this option to select an editor other than TED, such as XEDIT for CMS users.
- **Pictorial View** enables you to generate comments at the end of each Maintain procedure.

Terminal

If you select the *Terminal* option (or if you issue the EX MSETUP command at the FOCUS command prompt) the following dialog box is displayed:



If Winform elements such as check boxes, scroll bars, borders, and colors are not displaying correctly on your terminal, you can use the System Setup dialog box to configure the Winform facility for a different terminal or terminal emulator. You can see the effect of your selection by looking at the Example Window in the upper right corner of the screen, which shows what a check box and scroll bar look like using the terminal or emulator you have selected.

To select your terminal or emulator, place a check mark in the appropriate box and press the Enter key. When you are finished press PF4. If you wish to cancel what you have entered, press PF3.

The example window in the upper-right corner of the screen shows how the Painter will display special characters.

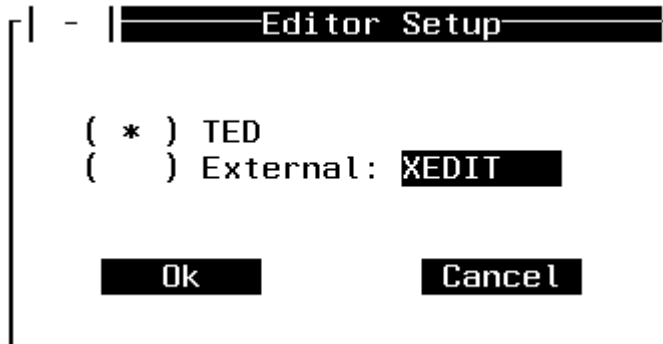
In addition, you may choose *Custom*. If you choose *Custom*, the Custom character set dialog box is displayed:

Custom character set				
Please, review following settings:				
	chr	dec	hex	apl
Left square bracket	[186	BA	[]
Right square bracket]	187	BB	[]
Up arrow	^	176	B0	[]
Down arrow	v	165	A5	[]
Press F1 to get mapping				
Ok PF4		Reset		

Initially your cursor is on the input area for the left square bracket. Position the cursor on the row of the character you wish to change. You can type a new value in the character column, or press PF1 to select from a list of characters. When you change a character, its decimal and hexadecimal values change automatically.

Editor

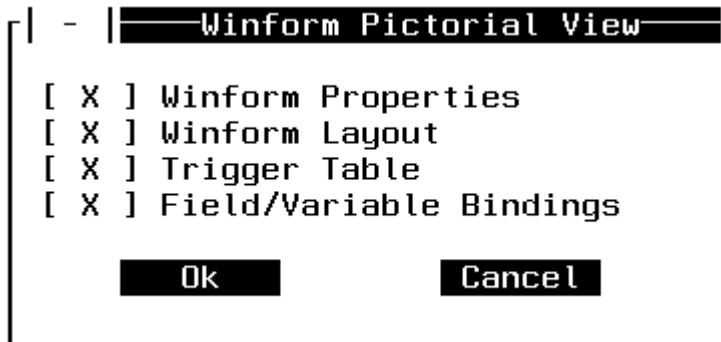
If you select the *Editor* option, the Editor Setup dialog box is displayed. From here you can select TED or—under CMS—specify another editor. (The external editor feature is not supported under MVS.) The editor you specify will be used when you edit the Maintain procedure. The dialog box looks like the following:



Press PF4 when you are finished selecting an editor. Press PF3 to cancel whatever you have entered.

Pictorial View

If you select the Pictorial View option the following dialog box is displayed:



By default the Painter generates four different types of comments at the end of each Maintain procedure:

Winform Properties	Includes information such as Winform name, title, and position.
Winform Layout	Displays how the Winform will look when it is executed.
Trigger Table	Displays PF key settings and their actions.
Field/Variable Bindings	Displays which stacks are being used in fields and grids.

If you do not want the Painter to generate comments, you can turn the feature off by clearing the appropriate check box.

Exit

Selecting *Exit* closes the Winform Painter and returns you to the FOCUS prompt. If you have made changes to the Winform file and have not saved them, you are prompted to save the changes.

Edit Menu

The Edit menu is used for customizing controls in the Winform. (Controls are called objects by the Winform Painter.) You may edit, move, copy, resize or delete a control. Selecting *Edit* yields:

_ Edit Object	F12
M ove	F5
C opy	
R esize	F24
D elete	F6

The Edit menu offers the following options:

- **Edit Object** enables you to edit a control in the Winform.
- **Move** enables you to move a control to another position in the Winform.
- **Copy** enables you to copy a control to another position in the Winform.
- **Resize** enables you to change the dimensions of a control in the Winform.
- **Delete** enables you to delete a control in the Winform.

Edit Object

The following controls may be edited. (Controls are called objects by the Winform Painter.)

- Fields
- Text
- Grids
- Browsers
- Buttons
- Check boxes
- List boxes
- Combo boxes
- Radio button groups

To edit a control in the Winform, either:

- Place the cursor on the control and press PF12.
- Select *Edit Object* from the Edit menu. The Painter prompts you to point to the control you wish to edit: do so and press the Enter key. If you do not wish to edit a control at this point, you may press PF3 to cancel the action.

The dialog boxes for editing controls are almost exactly the same as for creating controls. For more information on the control creation process, see *Objects Menu* on page 5-44.

Move

To reposition a control in the Winform:

1. Select *Move* or press PF5.
2. You are prompted to identify the control you wish to move.
Position the cursor at the control you wish to move and press the Enter key.
3. You are prompted to point where you want to move the control.
Position the cursor where you want to move the control and press the Enter key.
The control and its contents move to the new area.

Copy

To copy a control to another area in the Winform:

1. Select *Copy*.
2. You are prompted to point to the control you wish to copy.
Position the cursor at the control you wish to copy and press the Enter key.
3. You are prompted to point where you want to copy the control.
Position the cursor where you want to copy the control and press the Enter key.
The control is copied to the new area selected in the Winform.

Resize

To resize a grid, frame, or button:

1. Select *Resize* or press PF24.
2. You are prompted to point to the control you wish to resize. Position the cursor on the desired control and press the Enter key.
3. The control can be enlarged by moving a corner outward; it can be made smaller by moving a corner inward. You can move only the active corner, which by default is the lower right.

If the control has a border, you can make a different corner active by pressing PF10 or PF11. The Painter always highlights the border and identifies the active corner with an asterisk.

Resize the control by selecting a new location for the active corner: move the cursor to the desired position and press the Enter key. If the new position is not valid—for example, if it will overlap with another control—the Painter ignores your choice, giving you the opportunity to select a new position. Otherwise it resizes the control as you have specified.

Delete

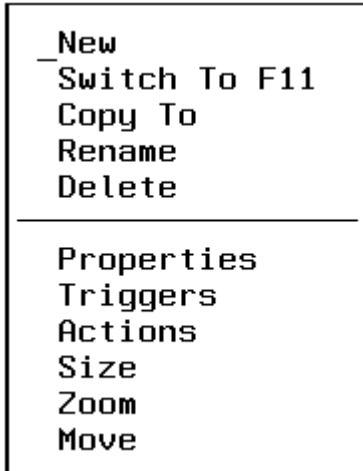
To delete a control in the Winform, either:

- Position the cursor on the control to be deleted and press PF6.
- Select *Delete* from the Edit menu. You are prompted to point to the control you wish to delete: position the cursor on the desired control and press the Enter key.

The control is deleted from the Winform.

Forms Menu

The Forms menu enables you to create and edit Winforms. You can edit the Winform you are currently working on, or you can create new Winforms in that same file. Selecting the Forms menu yields:



The Forms menu offers the following options:

- **New** enables you to create a new Winform while in an existing Winform.
- **Switch To** enables you to switch from one Winform to another.
- **Copy To** enables you to copy the current Winform to another Winform.
- **Rename** enables you to rename the current Winform.
- **Delete** enables you to delete the current Winform.
- **Properties** enables you to enter information about the Winform.
- **Triggers** enables you to update, add, or delete the Winform's form-level triggers.
- **Actions** enables you to update, add, or delete the Winform's system actions.
- **Size** enables you to resize the current Winform.
- **Zoom** enables you to return a Winform to full size.
- **Move** enables you to move the Winform to a different position on your screen.

New

Select *New* to create a new Winform while in an existing Winform.

Selecting *New* displays the Winform Properties dialog box shown in *Properties* on page 5-37.

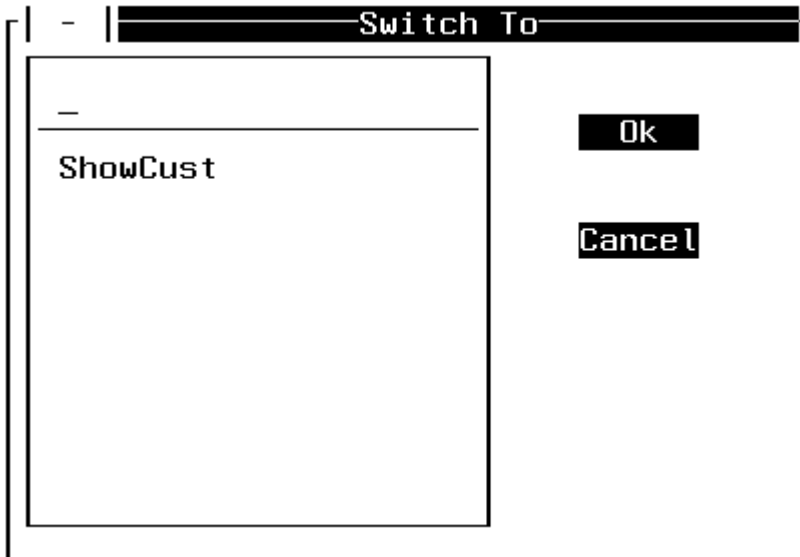
When you have finished entering the properties information, to:

- Confirm what you have entered, press PF4.
- Customize the colors of your Winform, press PF5. The Painter displays the Set Colors dialog box, which is described in *Colors* on page 5-41.
- Cancel the transaction, press PF3.

Switch To

Select *Switch To* in order to change from one Winform to another within a Winform file. You may also switch from one Winform to another by pressing PF11 while in a Winform.

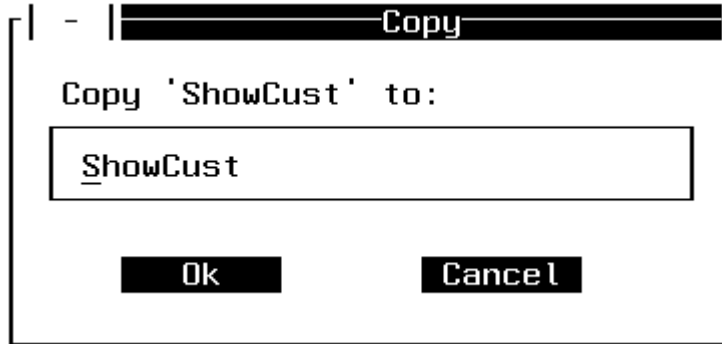
Selecting *Switch To* yields the Switch To dialog box which contains a list of all the Winforms in your current Winform file:



Enter the name of the Winform you would like to switch to and press PF4, or place the cursor under your selected Winform, and press the Enter key. This displays your selected Winform. If you place the cursor in the upper section of the box in the previous screen, and press PF4 or Enter, you can create a new Winform. Press PF3 to exit the Switch To screen.

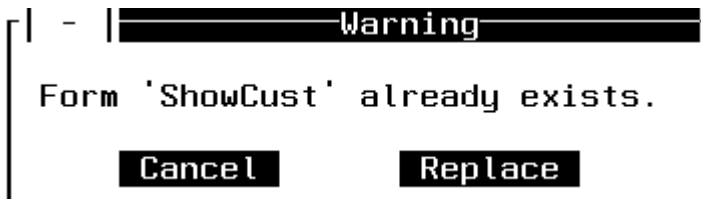
Copy To

Select *Copy To* to copy one Winform into another Winform within the same Winform file. Selecting *Copy To* generates the *Copy* dialog box:



The name of the current Winform is automatically supplied. Enter the name of the Winform you wish to copy an existing Winform to. Press the Enter key or PF4 to copy the Winform. Press PF3 if you wish to cancel the action.

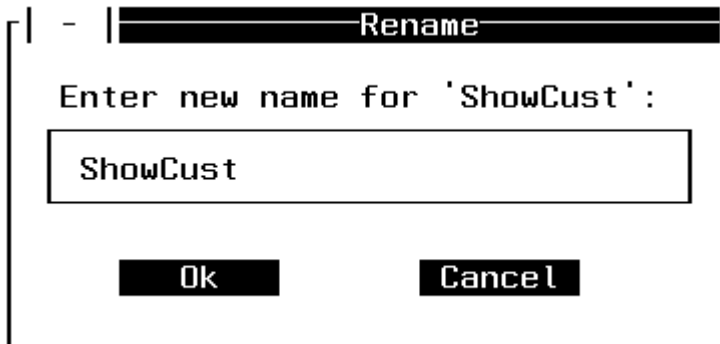
If you press PF4 or Enter, and the Winform already exists, the following dialog box is displayed:



If you wish to delete the Winform you are trying to copy to and replace it with the Winform you entered in the *Copy* dialog box, place the cursor under *Delete* and press the Enter key. If you wish to cancel the action, place the cursor under *Cancel* and press the Enter key.

Rename

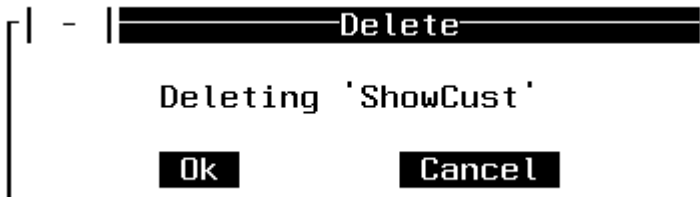
Select *Rename* to change the name of a Winform. Selecting *Rename* displays the Rename dialog box:



The name of the current Winform is automatically supplied. Enter the new name for the Winform and press the Enter key. Pressing PF3 cancels the action. If the name already exists, a dialog box similar to the warning dialog box in *Copy To* on page 5-35 is displayed.

Delete

Select *Delete* to erase a Winform. Selecting *Delete* displays the Delete dialog box:



Place the cursor under *OK* and press the Enter key if you wish to delete the Winform. If you select *OK* to delete the Winform, the Switch To menu appears minus the name of the current Winform. If you do not wish to delete the Winform, place the cursor under *Cancel* and press the Enter key.

Properties

Select *Properties* to enter information about the Winform. The following is the Winform Properties dialog box before information is entered into it:

The screenshot shows the Winform Properties dialog box with the following fields and controls:

- Menu: File Edit Forms Objects Cases Help < FOCEXEC >
- Title bar: Winform Properties
- Winform title: [Empty text box]
- Winform name: [Empty text box]
- Options: Pop-up Border
- Stacks list: [Empty list box]
- Buttons: Add to Both, Add to Source, Add to Dest, Delete F6
- Source Stacks table:

Name	Start at Focindex/First Row
[Empty]	[Empty]
- Destination Stacks table:

Name	Refresh to Focindex/Prev Row
[Empty]	[Empty]
- Bottom buttons: Ok F4, Colors F5, Cancel F3, Help F1

The following shows the Winform Properties dialog box after the stkcust and stkrent stacks have been specified as source and destination stacks:

The screenshot shows the Winform Properties dialog box with the following fields and controls:

- Menu: File Edit Forms Objects Cases Help < FOCEXEC >
- Title bar: Winform Properties
- Winform title: Customer Video Rental History
- Winform name: ShowCust
- Options: Pop-up Border
- Stacks list:
 - stkrent
 - stkcust
 - stkrent
- Buttons: Add to Both, Add to Source, Add to Dest, Delete F6
- Source Stacks table:

Name	Start at Focindex/First Row
stkcust	[X]
sktrent	[X]
- Destination Stacks table:

Name	Refresh to Focindex/Prev Row
stkcust	[X]
sktrent	[X]
- Bottom buttons: Ok F4, Colors F5, Cancel F3, Help F1

Winform Title

The Winform title is text that will be displayed on the screen in the Winform's title area, centered at the top of the Winform, if a Winform has a border. If a Winform does not have a border the title will not be displayed. Supplying a title is optional.

Winform Name

Each Maintain procedure can display many different forms, but all of the forms are stored in one file. Therefore, in order to be able to paint or display a specific Winform, all of the forms must be named within the file. In order to coordinate the Maintain procedure and the Winforms, both the Maintain procedure logic and the Winform information must have the same file name, but the file type (or ddname) must be either FOCEXEC or MAINTAIN for the processing logic or Winforms for the screen information. The Winform name is required.

In this dialog box, the Winform name is the name that you will use in the Maintain procedure to reference the Winform you are building now. For instance, if you supply a name of ShowCust for the Winform, in your Maintain procedure you could display the Winform by supplying the following line:

```
WINFORM SHOW ShowCust
```

Pop-up

By default, *Pop-up* is checked which means that after the user exits the form, it will no longer be displayed on the screen. If *Pop-up* is not checked, when the user exits the form, it is still displayed to the user, but it is not active. This is true as long as the user is still viewing a Winform. If you invoke a function in Winform1 that displays Winform2, and you exit Winform2, one of the following happens:

- If *Pop-up* is checked Winform2 is no longer displayed on the screen.
- If *Pop-up* is not checked, Winform2 is still displayed on the screen but it is not active. This means that you cannot perform any action on that Winform.

Border

Border is used to put a border around a Winform. It is optional, but must be selected if you want to display a Winform title or Control box menu. The default is set to on.

Stacks

A stack is a simple table: every stack column corresponds to a data source or calculated field, and every stack row corresponds to a data source record (a path instance). The stack itself represents a data source path. You can populate a stack by retrieving data from a data source, calculating values, or copying all or part of an existing stack.

The Current Area is Maintain's unnamed default stack, and has one row.

Winforms do not display data directly from a data source or directly update a data source; they display data from, and write data to, stacks or the Current Area. These are known as the source and destination stacks. For each Winform, you can use as many source and destination stacks as you wish.

You can select any of the stacks in the Stacks combo box as a source or destination stack for every field, browser, and grid that you create in all the Winforms in the Winform file. Simply select the desired stack, and then click the appropriate Add button to the right of the list. Depending upon the button you clicked, the stack name is copied into the Source Stacks and/or Destination Stacks drop boxes at the right of the dialog box.

You must select the same stacks for both source and destination.

Source Stacks

For each stack in the Source Stacks list, you can check the *Start at FocIndex* check box, which determines the current row—that is, the current position within the stack—when the Winform is opened. If you design the Winform:

- With the Start at FocIndex attribute, when the Winform is opened the stack starts out with the same position it had just prior to the Winform being opened. This ensures that the stack's position is consistent inside and outside the Winform.

Maintain accomplishes this by using the system variable FocIndex to determine the current row. This enables you to retain the stack's position when you open the Winform, and makes it possible for you to dynamically manipulate the current row by assigning a value to FocIndex.

- Without the Start at FocIndex attribute, when the Winform is opened the stack's current position is the first row, regardless of where it was prior to the Winform.

If you do not specify any source stacks, the source defaults to the Current Area.

Destination Stacks

For each stack in the Destination Stacks drop box, you can also check the *Refresh to FocIndex* check box, which controls the Winform's behavior when someone invokes a trigger that interrupts—and later returns control to—the Winform. When control returns to the Winform, it refreshes the data that it was displaying in case the stack had been updated in the interim. If you design the Winform:

- With the Refresh to FocIndex attribute, when the Winform refreshes its data from the stack it will also refresh its position within the stack. This ensures that the Winform reflects the most recent changes not only to the stack's data, but also to its position. It accomplishes this using the system variable FocIndex to determine the current stack row.

For example, if a trigger manipulates the stack and changes the current position—say, if the trigger calls a second Winform that also displays that stack, and a user moves to another row—Maintain retains that new stack position when you return to the original Winform. This also makes it possible for you to dynamically manipulate the current row by assigning a value to FocIndex within the intervening trigger.

- Without the Refresh to FocIndex attribute, the current row is unchanged by anything that happened during the trigger.

If you do not specify any destination stacks, the destination defaults to the Current Area.

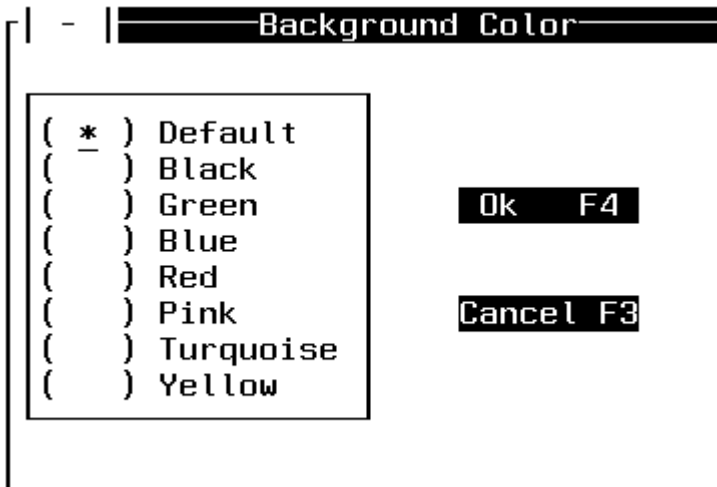
Stack Buttons

There are four buttons in the Winform Properties dialog box that allow you to perform actions on the stacks in the Winform.

- **Add to Both** adds the selected stack to both the Source Stacks and Destination Stacks lists. This is the default button.
- **Add to Source** adds the selected stack to the Source Stacks list.
- **Add to Dest** adds the selected stack to the Destination Stacks list.
- **Delete** removes the selected stack from the current list box, that is, from the list box in which the cursor is located.

Colors

You can choose a background color for a Winform by selecting *Colors*. This displays the Background Color dialog box:



Check the desired background color and click *OK*, or click *Cancel* to exit the Background Color dialog box without saving any changes.

You can also change the background color dynamically at run time by issuing the WINFORM SET command in the Maintain procedure, as described in *WINFORM* in Chapter 7, *Command Reference*.

Triggers

Select Triggers to assign or change form-level triggers. Form-level triggers are triggered when an end user presses the specified function key when the cursor is anywhere on a form except for a spot occupied by a control. For more information about specifying triggers, see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Actions

Select Actions to assign or change system actions. System actions are similar to form-level triggers, but instead of invoking functions, they invoke special system-defined actions that do things like close the current form or exit the current procedure. For more information about specifying actions, see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Size

Select Size to reduce or enlarge the size of the Winform. Selecting *Size* generates a screen similar to the following:

Screen ___ of ___

Customer ID : ___

Name : _____

Address : _____

Phone : _____

Movie Code	Copy	Return Date	Fee
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

stkrent

Prev Cust PF1 Next Cust PF2 Done PF4

-----1-----2-----3-----4-----5-----6-----7-----*

F1=Help 2=AddLn 3=Exit 4=NewFld 5=Move 6=DelObj 7=Up 8=Down 9=DelLn 12=EdtObj

A solid border is generated around the Winform. The cursor is positioned at the bottom right-hand corner of the Winform. To reduce or enlarge the Winform, move the cursor to where you would like the new bottom right-hand corner to be and press the Enter key. If, while changing the size of the Winform, you eliminate a control, the following dialog box is displayed:

Warning

Some objects will be deleted

Ok Cancel

If you do not wish to remove the control from the Winform, place the cursor under *Cancel* and press the Enter key. The Winform is returned to its original size. If you want to accept the new size of the Winform, place the cursor under *OK* and press the Enter key. The Winform is saved as it appears and the controls are deleted.

Zoom

Select Zoom to restore a Winform to full-size. Use this option if the Winform has been previously reduced with the Size option.

Move

Select Move to move the Winform to a different position on your screen. This can be useful if you wish to clear part of the screen for additional controls such as a button. Selecting *Move* generates a screen similar to the following:

```

*
Screen  ___ of  ___

Customer ID : ___

Name       : _____
Address    : _____
Phone     : _____

          stkrent
  Movie Code  Copy  Return Date  Fee
  ---
  ---
  ---

Prev Cust PF1  Next Cust PF2  Done PF4

-----+-----1-----+-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----7-----+-----
F1=Help 2=AddLn 3=Exit 4=NewFld 5=Move 6=DelObj 7=Up 8=Down 9=DelLn 12=EdtObj

```

A solid border is generated around the Winform, and the cursor is positioned at the upper left-hand corner of the Winform. Move the cursor to where you would like the new upper left-hand corner to be and press the Enter key. The Winform is shifted to its new position.

Objects Menu

The Objects menu enables you to create controls for your Winform. (Controls are called objects by the Winform Painter.) Selecting the *Objects* menu yields:

_Field	F4
_Text	F13
Grid	F14
Browser	F15
Frame	F16
Button	F17
Checkbox	F18
Listbox	F19
Combobox	F20
Radio Group	F21
<hr/>	
Gen Segment	
Gen Master	

The Objects menu offers the following options:

- **Entry Field** enables you to display a field in the Winform.
- **Text** enables you to add text to a Winform.
- **Grid** enables you to place a grid in the Winform.
- **Browser Functions** enable you to place a browser in the Winform.
- **Frame** enables you to place a frame around controls in the Winform.
- **Button** enables you to place a button in the Winform.
- **Checkbox** enables you to place a check box in the Winform.
- **List Boxes** enables you to place a list box in the Winform.
- **Combobox** enables you to place a combo box in the Winform.
- **Radio Group** enables you to place a radio button group in the Winform.
- **Gen Segment** enables you to place all of a segment's fields into the Winform at one time.
- **Gen Master** enables you to place all of a data source's fields into the Winform at one time.

Entry Field

An entry field enables a user to enter and edit a simple variable. When a Winform is displayed, the entry field's initial value comes from the current row (indicated by FocIndex) of the specified stack column, from a default value, or from the Current Area. When the Winform is closed, the field's value is written to the current row of the specified stack.

For general information about how an end user manipulates an entry field, see *Using Entry Fields* on page 5-3.

Procedure How to Place an Entry Field in a Winform

1. Select *Field* from the Objects menu, or press PF4.

You are prompted to point to where you would like to place the field in the Winform.

2. Move the cursor to the position where you wish to display the field and press the Enter key, or press PF3 to cancel the action.

The Create Field dialog box is displayed:

When you are done making your selections in the Create Field dialog box, you may either:

- Press PF4 to create the field and return to the Winform Painter.
- Press PF3 to cancel field creation and return to the Winform Painter.

Field Combo Box

The Field combo box lists all the fields in the Master Files that are used by the current procedure. Select a field to associate with this entry field in the Winform, or type the name of a user-defined variable (that is, a variable created with the COMPUTE or DECLARE commands) to associate with this entry field.

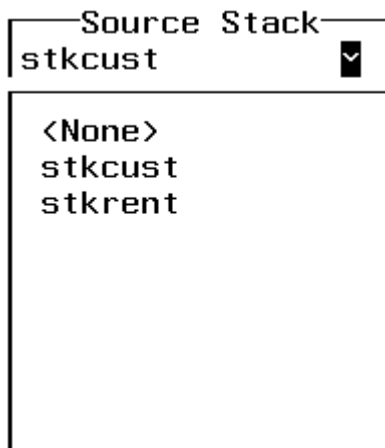
If you specify a user-defined variable, the procedure must declare it before displaying the Winform.

Tip: If you want to add all of the fields in a data source or all of the fields in one segment of a data source to a Winform quickly, use the Gen Segment or Gen Master commands. For more information, see *Gen Segment* on page 5-75 and *Gen Master* on page 5-77.

Source

For the source of the field's data when the Winform is first displayed, you may select:

- **Default.** If you check *Default* you must enter a value in the Default value box. Default is not supported in the current release.
- **Current Area.**
- **Stack.** You may use any stack contained in the Source Stack drop box. To view the available stacks, position the cursor in the *Source Stack* entry field and press the Enter key. The Source Stack drop box is displayed:



Position the cursor on the stack you wish to use and press the Enter key. You are returned to the Create Field dialog box with your selection displayed in the Source Stack dialog box.

Destination

You may use any stacks contained in the Destination Stack drop box. To view the stacks that are available to you, place the cursor in the *Destination Stack* entry field and press the Enter key. A drop box similar to the Source Stack list box is displayed, containing the stacks available to you. Position the cursor over the stack you wish to use and press the Enter key. You are returned to the Create Field dialog box with your selection displayed in the Destination Stack drop box.

You must specify the same stack for both source and destination.

Object Name

Provide a name for the field, or accept the default name. Names enable you to refer to controls within your procedure, so that you can dynamically manipulate them in response to run-time events.

Default Value

Enter a default value if you wish to use the same value in many instances. For example, if your data source contains a field named *City*, and *City*'s most frequent value will be "New York City", you may wish to specify this as the default value. If you specify a default value you must check *Default* as your source.

The Winform Painter does not support default values for fields in the current release.

Length

Length specifies the display length of the field in the Winform. It defaults to the length needed to display the field's data and display options as defined in the Master File or in the COMPUTE or DECLARE command.

- **Numeric and date fields.** When an application user enters data into a field, he or she can use any display options (such as separators and translation in date fields, or commas in double precision floating point fields) that are valid for that field's data type. Once the user presses Enter or a function key, the Winform displays that data using only the display options that are actually defined for the field in the Master File or in the COMPUTE or DECLARE command. If you specify a length that is shorter than the default, the Winform does not show any display options.

The display length must accommodate the largest value to be displayed or entered. To ensure this, we recommend that you specify a length at least as great as the length of the data as defined in the Master File or in the COMPUTE or DECLARE command. You can specify a greater length if you wish to accommodate display options.

- **Alphanumeric fields.** If you specify a display length that is less than the default, and a value exceeds the display length, the application user can enter or display the entire value by scrolling the field using the LEFT and RIGHT system actions (which are the default settings of the PF10 and PF11 keys respectively). This can be helpful if you need to keep Winform fields short in order to conserve screen space.

As elsewhere in FOCUS, display options are significant only when displaying or printing a value. When Maintain writes a field from a Winform to a stack or the Current Area, it writes the data only, not the display options.

Prompt

Prompt enables you to display a field with more descriptive text if it is deemed necessary. For example, if you wish to display a field called Partno in a more meaningful way, you might use Part Number. *Prompt* is checked by default.

Uppercase

Check Uppercase if you wish to display a field in uppercase. If *Uppercase* is not checked, the field displays as it was originally entered, whether uppercase, lowercase or mixed case. *Uppercase* is checked as the default.

Protected

Check Protected if you want to deny the application user the ability to change the value of a field. This is useful for protecting key fields. *Protected* is unchecked by default.

Accepts

The Accepts button allows you to specify data validation values for the current field. At run time, if an end user attempts to enter invalid data into the field on a Winform, the invalid data is not accepted, and the field's valid values are displayed. You can specify the valid values in:

- The field's ACCEPTS attribute in the Master File. At run time, values specified here take precedence over values specified in the Winform Painter.

The existence of an ACCEPTS attribute automatically enables data validation for that field in a Winform; you do not need to set anything in the Accept List dialog box.

- The Winform Painter's Accept List dialog box. At run time, values specified here are ignored if values exist for the field in an ACCEPT attribute in the Master File.

To activate data validation for the field using values specified in the Winform Painter, ensure that the field has no ACCEPTS attribute in the Master File, select the *Stack List* radio button in the Accept List dialog, and select one or both of the *On Error* and *On PFKey* check boxes in the Accept List dialog.

Because data validation values in the Master File take precedence over data validation values specified in the Winform Painter, if values for a field already exist in the Master File, the Winform Painter prevents you from specifying values for that field in the Painter.

To edit data validation information in the Winform Painter for the current field, press PF6. The Accept List dialog box is displayed:

```

- | Accept List
[ ] On Error      ( * ) Off
[ ] On PFKey     ( ) Stack List
                  ( ) Range
                  Lower: 
                  Upper: 
                  Stack name: 
                  Value column: 
                  Display column: 
                  Ok F4      Cancel F3
  
```

You can edit the following data validation properties:

- **On Error.** By checking *On Error* when *Stack List* is selected, if an end user enters invalid data, Maintain displays a list of valid values from which the end user can select.
- **On PFKey.** By checking *On PFKey* when *Stack List* is selected, if an end user presses a predefined function key while the cursor is on this field, Maintain displays a list of valid values from which the end user can select. PF1 is the default key for the Accepts system action; you can switch to a different function key using the Actions option of the Forms menu.
- **Validation status.** The following radio buttons control the field's data validation status:
 - **Off.** *Off* should be checked if you do not wish to assign data validation tests in the Winform for this field. You may still use Accepts in the Master File.
 - **Stack list.** If you check *Stack List*, and *On Error*, *On PFKey*, or both are checked, and the application user enters an invalid value, a list of valid entries is displayed. When you check *Stack List*, you must also enter a stack name and display column that contains the acceptable values.
 - **Range.** *Range* is used to define an acceptable range of values. To the right of *Lower*, type the lowest value of the range. To the right of *Upper*, type the highest value of the range. You may use numbers or letters. *Range* is not supported in the current release.
- **Stack name.** If you are using a list to define the valid values, type the name of the stack that contains these values.

If the valid values are meaningful to the application user, you can identify a single stack column to contain the list of values and to display that same list on the screen. However, if the values need to be explained, you can use one column to contain the list of valid values and another column to contain a description of each value: Maintain displays only the description column on the screen, and when the application user picks an item in the description column, Maintain writes the corresponding item from the valid data column to the field.

For example, in an invoice application you might use the ValidData stack to contain a list of valid product codes. You can keep the product codes themselves in the ValidData.Code column, and the product name corresponding to each code in the ValidData.Name column. When a user presses PF1 to see a list of valid values, and then chooses "Hammer" from the ValidData.Name column, Maintain copies the Hammer product code—11375—from the ValidData.Code column to the field.

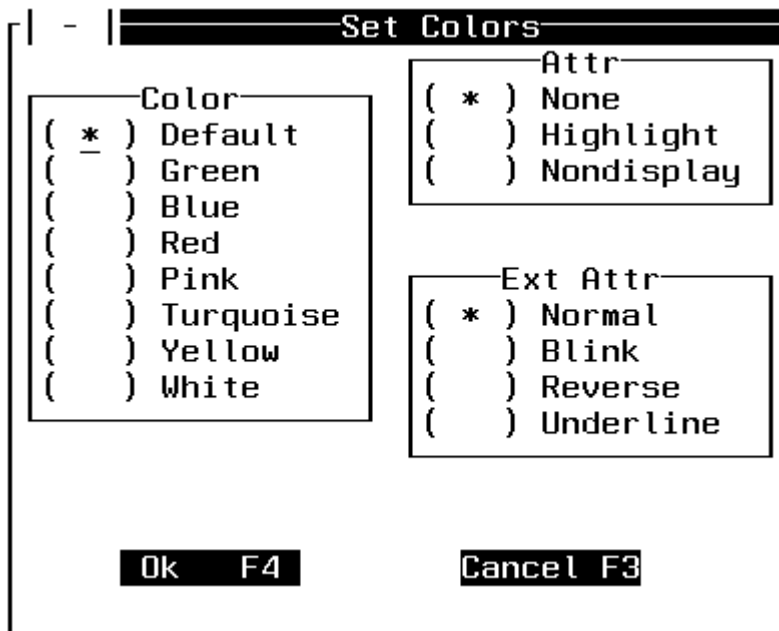
- **Value column.** If you specify a stack to contain a list of valid values, you must supply the name of the column that you want the application user to see displayed on the screen.
- **Display column.** If you specify a stack to contain a list of valid values, and you are using two columns—one to provide the actual values, and the other to display the descriptions—then supply the name of the display column here. Otherwise, if you wish to use one column both to contain the values and display them, leave this field blank and it defaults to the column name you supply for *Value column*.

Triggers

You may assign up to twenty-five control-level triggers to each entry field. These triggers are invoked when an end user presses the specified function key when the cursor is in the entry field. For more information about triggers, see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Color

Color displays the Set Colors dialog box, which allows you to change the color of the field:



In the Set Colors dialog box, you may change the color, attributes, and extended attributes of the field. The current settings are indicated with asterisks. Note that you can also change the color of a Winform control dynamically at run time by issuing the WINFORM SET command from the Maintain procedure, as described in *WINFORM* in Chapter 7, *Command Reference*.

In the Color radio button group, you may change the color of the fields in your Winform by marking the appropriate color. It is initially set to the default color. The color is dependent on your hardware and terminal emulation. Your terminal must be capable of displaying color to use this feature.

In the Attr radio button group, you may:

- Bold a field or its text by marking *Highlight*.
- Select *Nondisplay* to hide the information you are entering. This feature can be used as a security measure.

In the Ext Attr radio button group, you may:

- Select *Blink* to have either the selected field or its text blink.
- Select *Reverse* to reverse the color background of the selected field or its text.
- Select *Underline* to underline the selected field or its text.

In order to use color, attributes, or extended attributes, your terminal or terminal emulator software must support it. When you are finished, press PF4. Press PF3 if you wish to cancel what you have entered.

Text

The Text control enables you to place static text into a Winform, such as instructions or a title.

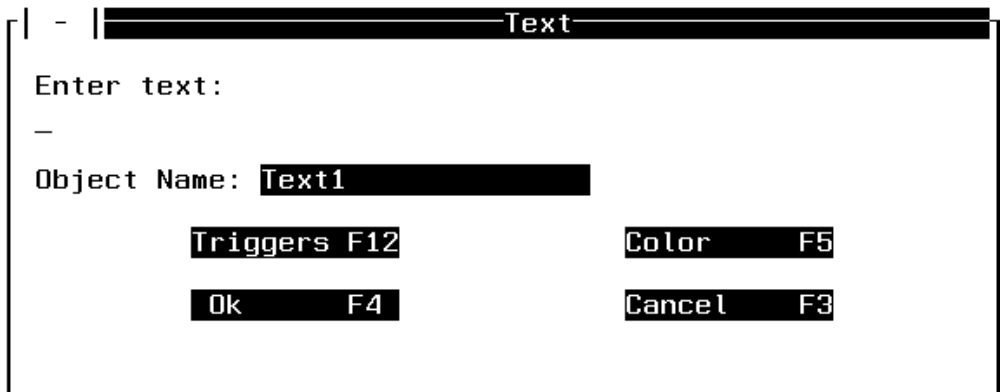
Procedure How to Place Text in a Winform

1. Select *Text*.

You are prompted to point to where you would like to place the text in the Winform.

2. Move the cursor to the position where you wish to display the field and press the Enter key or PF4, or press PF3 to cancel the action.

If you choose to display the field, the Text dialog box is displayed:



Enter the text as you would like it to be displayed.

Object Name

Provide a name for the text, or accept the default name, so that you can refer to the text within your procedure. Text1 is the initial default.

Triggers

You may assign up to twenty-five control-level triggers to each text control. These triggers are invoked when an end user presses the specified function key when the cursor is in the text control. For more information about triggers see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Color

Press PF5 if you wish to edit the color of the text. Pressing PF5 yields the Set Colors dialog box. For detailed information about the Set Colors dialog box see *Color* on page 5-51.

Grid

A grid is a stack editor that enables you to display and edit selected stack columns.

Procedure How to Place a Grid in a Winform

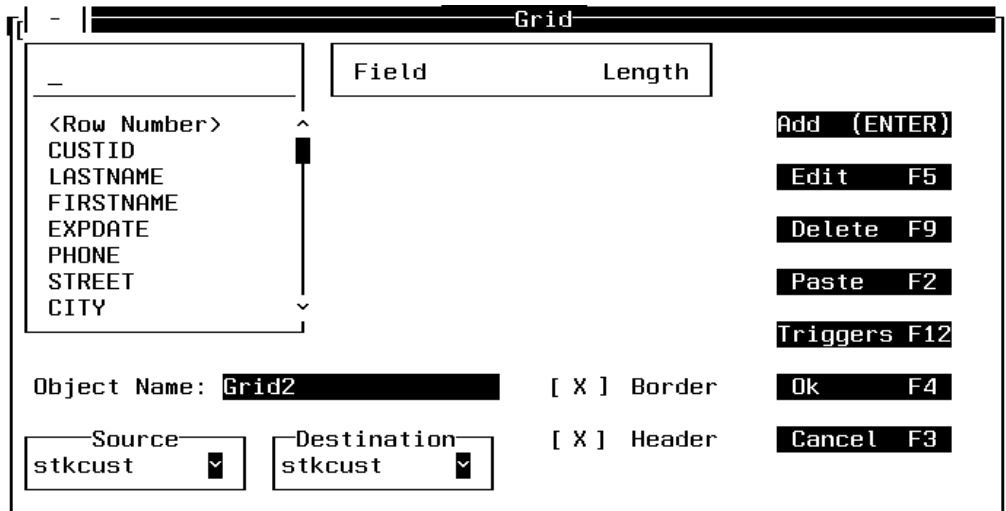
1. Select *Grid*.

You are prompted to point to the top left corner of the grid and press the Enter key, or press PF3 to cancel.

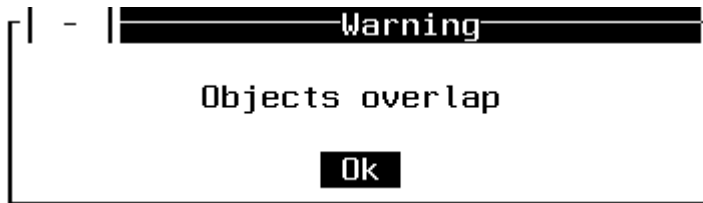
Move the cursor to the position where you wish the upper left-hand corner of the grid to be displayed, and press the Enter key.

2. You are prompted to point to the bottom right corner of the grid.

Move the cursor to the position where you wish the lower right-hand corner of the grid to be displayed, and press the Enter key. The Grid dialog box is displayed:



If the grid you are trying to put in the Winform covers some of the same space as another control, the following is displayed:



You must change the position of your grid so that it does not overlap with other Winform controls. Press Enter and you are returned to the prompt for the lower right-hand corner of the grid. When you choose a proper place to put the grid, the Grid dialog box is displayed.

When you are finished working on the Grid dialog box, press PF4 to save the information or PF3 to cancel the action.

Grid Columns

The combo box in the left part of the Grid dialog box displays the fields in the Master File. As you add fields to the grid, another list box is created below Field in the Grid dialog box. The length of the field comes from the Master File and may be changed. Row Number allows you to see the row in the stack that is visible to the user. It is a good idea always to include it. You may not change the value of Row Number.

Add

Select Add to add columns from the source stack to a grid.

In order to add a field to the grid, you can place the cursor under the name of a field in the left combo box and press the Enter key. You may change the length of the field in the grid by changing it in the length column.

Edit

Select Edit to edit the column name and to specify other column attributes of a field that is currently displayed in the grid. To edit a column name, press PF5. The Edit Column dialog box is displayed:

The Edit Column dialog box shows the column name last selected or edited in the grid, in this case MovieCode. You may edit the following column properties:

- **Title.** You can edit the column's title to another title of your choosing.
- **Length.** You can edit the field's length (that is, the column's width) by placing the cursor at the number following *Length* and entering the new length. For further details concerning column width, see *Length* on page 5-48.
- **Uppercase.** You can make the column's contents uppercase by checking *Uppercase*. The default is mixed-case.
- **Protected.** You can protect the column from being edited by checking *Protected*. The default is unprotected. This feature should be used with key fields to prevent a key field from being updated.
- **Accepts.** You can specify a data validation test by pressing PF5. A dialog box is displayed that allows you to add Accept data validation tests. You may only add Accepts to fields that do not have Accepts defined in the Master File. The Accepts dialog box is described in *Accepts* on page 5-49. Accepts is not supported for grid columns in the current release.

- **Color.** You can choose colors for the column by pressing PF6. The Set Colors dialog box is displayed. The Set Colors dialog box is described in *Color* on page 5-51. Color is not supported for grid columns in the current release.

You can also select another field to edit. To do so, place the cursor under the selected field and press the Enter key. Scroll through the combo box to see which other fields you can edit. Press PF8 to scroll forwards or PF7 to scroll backwards.

You can exit the Edit Column dialog box, saving what you have entered, by pressing PF4. This returns you to the Grid dialog box. Otherwise you can cancel the action and return to the Grid dialog box by pressing PF3.

Delete

Select Delete to move a column to another part of the grid or erase a column from the grid. Position the cursor in the right list box at the column you wish to delete, and press PF9. The column name is removed from the list of displayed column names, and the column is removed from the grid.

Paste

Select Paste to reposition a column in the grid that has previously been deleted. Press PF2 to paste a column. This brings the column name back to the active column list box in the Grid dialog box, and returns it to the grid.

Triggers

You may assign up to twenty-five control-level triggers to each grid. These triggers are invoked when an end user presses the specified function key when the cursor is in the grid. For more information about triggers, see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Border

Select Border to put a border around a control. The default is set to on.

Header

Select Header to include column titles in the grid. The default is set to on.

Source

Source displays all source stacks and lets you select one as the source of your grid's data.

Destination

Destination displays all destination stacks and lets you select one as the destination of your grid's data.

You must specify the same stack for both source and destination.

Object Name

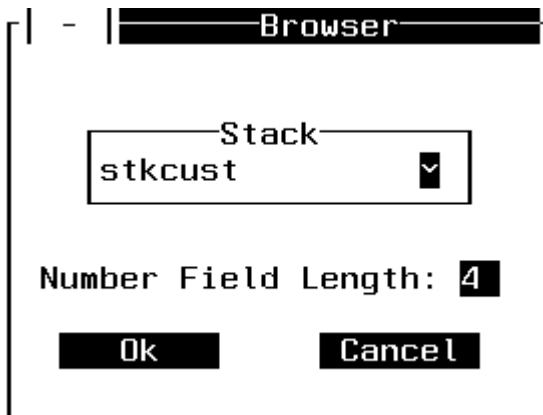
Provide a name for the grid, or accept the default name, so that you can refer to the grid within your procedure.

Browser Functions

The browser generates the necessary logic and Winform controls to enable you to display one stack row, and to scroll forward and backward through the stack one row at a time. The browser adds two buttons for moving forward and backward through the stack and changing the value of FocIndex accordingly. The browser also adds text and two fields—the first FocIndex, the second FocCount—in the upper-left corner of the Winform to identify the current row to the user. There can only be one browser per application.

Procedure How to Place a Browser in a Winform

1. Select *Browser*. The Browser dialog box is displayed:



2. When you are finished entering information in the Browser dialog box press PF4 to save the information, or PF3 to cancel the transaction.

Stack

Select the stack that you would like to scroll through. All stacks defined to the Winform are displayed.

Number Field Length

The Number Field Length field is used to specify a length for the field that is displayed to indicate how many rows were retrieved, and which row you are currently positioned on.

Frame

Select Frame to put a border around an area in the Winform. A useful feature of the frame is the ability to move all the controls in the frame at one time, simply by moving the frame itself.

Procedure How to Place a Frame in a Winform

1. Select *Frame*.

You are prompted to place the cursor at the upper left-hand corner of the frame you wish to create.

Place the cursor at the desired upper left-hand corner of the frame, and press the Enter key.

2. You are prompted to place the cursor at the lower right-hand corner of the frame you wish to create.

Place the cursor at the desired lower right-hand corner of the frame, and press the Enter key.

A screen similar to the following appears after you have finished positioning the frame, in this case around Customer ID, Name, Address and Phone:

The screenshot shows a window titled "Customer Video Rental History" with a status bar "Screen ___ of ___". Inside the window, a rectangular frame encloses the following elements:

- Customer ID : ____
- Name : _____
- Address : _____
- Phone : _____

Below the frame, there is a table with the following structure:

stkrent			
Movie Code	Copy	Return Date	Fee
—	—	—	—
—	—	—	—
—	—	—	—

At the bottom of the window, there are three buttons: "Prev Cust PF1", "Next Cust PF2", and "Done PF4".

Button

When a user clicks a button to which you have assigned a trigger, it immediately executes the associated Maintain function or system action. (Maintain functions—often simply called functions—are referred to in the Winform Painter as cases.) For example, clicking a button might trigger a function that updates a data source. The user selects a button by moving the cursor to the desired button and pressing Enter.

For general information about how an end user would manipulate a button, see *Using Command Buttons* on page 5-7.

Procedure How to Place a Button on a Winform

1. Select *Button*.

You are prompted to place the cursor at the upper left-hand corner of the button you wish to create.

Place the cursor at the desired upper left-hand corner of the button, and press the Enter key.

2. You are prompted to place the cursor at the lower right-hand corner of the button you wish to create.

Place the cursor at the desired lower right-hand corner of the button, and press the Enter key.

The Button dialog box is displayed after you have finished positioning the button:

The screenshot shows a dialog box titled "Button" with the following fields and options:

- Text:** A text entry field containing "Prev Cust PF1".
- Justification:** Three radio button options: "Left", "Center" (which is selected), and "Right".
- Trigger:** A text entry field containing "prevcust" and a button labeled "Cases F5".
- Shortcut key (PF1-PF24):** A text entry field containing "PF1" and a button labeled "PFKeys F6".
- Default Button:** A checkbox that is currently unchecked.
- Border:** A checkbox that is currently unchecked.
- Object Name:** An empty text entry field.
- Buttons:** "Ok F4" and "Cancel F3" buttons at the bottom.

When you have finished entering information about a button, press the *OK* button. The button is then displayed in the Winform. Press PF3 if you wish to cancel the action.

Text

Enter into the Text entry field what you wish to appear on the button. The size of the entry field is dependent on the dimensions you have chosen for the button. This means that you can only enter text that fits on the button.

Justification

Justification allows you to choose whether to center, left-justify or right-justify the text on the button. Place an X next to your choice of justification.

Trigger

You may assign one Maintain function or the Exit or Close system action to the button. This function or action is invoked when an end user presses the specified shortcut key, or the Enter key, if *Default Button* is selected. For more information about triggers, see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Shortcut key

Enter a PF key number if you want to be able to press a PF key as a shortcut, rather than having to move the cursor to the button and press the Enter key. To see a list of PF keys and the triggers currently assigned to them, press PF6 or move the cursor to the *PFKeys* button and press the Enter key. When running the application, pressing the PF key or moving the cursor to the button and pressing Enter produces the same results.

Tip: You may want to avoid assigning the Enter key as a button's shortcut key. Because end users must press the Enter key to click a button, it is recommended that you do not assign the Enter key as a trigger unless you intend end users to trigger the specified function or system action each time they click the button.

Default Button

If Default Button is selected, and the user runs the application, pressing Enter clicks the button. In other words, Enter serves as a shortcut key for the button. There can only be one default button per form.

Border

Check Border to put a box around the button.

Object Name

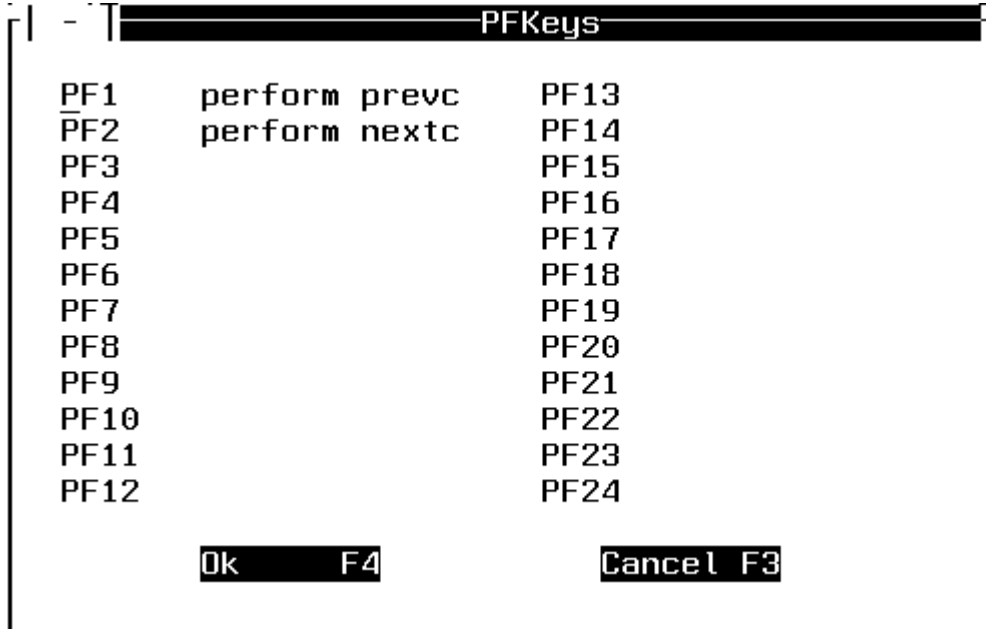
Provide a name for the button, or accept the default name, so that you can refer to the button within your procedure.

Cases

Select Cases to see the functions and system actions available for use as a trigger action. The Available Cases dialog box is displayed. For more information about the Available Cases dialog box, see *Specifying Triggers* on page 5-85.

PFKeys

Select PFKeys to see which PF keys currently have form-level triggers assigned to them. You can use this dialog box to select an available key (that is, one to which no form-level triggers or button shortcuts are assigned) for use as the current button's shortcut key. To select an available key, place the cursor under the PF key you wish to use and press PF4.



To exit the PFKeys dialog box without assigning a shortcut key, press PF3 or move the cursor to the *Cancel F3* button and press the Enter key.

Note that you cannot use this dialog box to change existing key assignments; it is designed for displaying key assignments only.

Checkbox

A check box enables the end user to select or clear (that is, deselect) an option. When you design the check box, you associate a variable with it; at run time, if the end user checks the box, the variable is assigned a value of 1; if the end user clears the check box, the variable is assigned a value of 0.

You can test the value of the variable in your application after the form is displayed to determine whether or not the box is checked. If you wish a check box to be checked by default, you can assign a value of 1 to the check box variable at run time before displaying the Winform.

For general information about how an end user would manipulate a check box, see *Using Check Boxes* on page 5-5.

Procedure How to Place a Check Box on a Winform

1. Select *Checkbox*.

You are prompted to point to a location and press the Enter key or PF3 to cancel.

2. Place the cursor at the desired location and press the Enter key or press PF3 to cancel.

If you press the Enter key, the Check Box dialog box is generated:

The screenshot shows a dialog box titled "Check Box". It contains the following elements:

- Text:** A text input field.
- Variable:** A text input field.
- Object Name:** A text input field.
- Define variable as format I1
- Triggers F12** button
- Ok F4** button
- Cancel F3** button

Text

Enter what you wish to appear next to the check box.

Variable

This is the name of the variable that will be associated with the check box. At run time, if the end user checks the box, the variable is assigned a value of 1; if the end user clears the check box, the variable is assigned a value of 0. You can test the value of the variable in your application after the form is displayed to determine whether or not the box is checked, and you can assign a value to the variable before the form is displayed to provide a default setting for the check box.

The application must declare the field before it is referred to in a Winform. Enter the name of the variable as it will appear in your source code. For further information about naming conventions, see Chapter 6, *Language Rules Reference*.

Object Name

Provide a name for the check box, or accept the default name, so that you can refer to the check box within your procedure.

Define Variable as Format I1

Check this box if you want Maintain to declare the variable automatically. Maintain declares the variable in your WINFORMS file as a one-digit integer (a format of I1), therefore you should not define this variable yourself in your application.

Triggers

You may assign up to twenty-five control-level triggers to each check box. These triggers are invoked when an end user presses the specified function key when the cursor is in the check box. For more information about triggers see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

List Boxes

List boxes display a list of items from which the end user can select one item. The list is taken from a stack column at run time when the Winform is displayed; you identify which stack column to use when you design the Winform.

If there are more list items than can fit in the list box, automatic scroll bars enable the application user to scroll through the list.

When the end user selects an item from the list, `FocIndex` points to the stack row that the user selected. For example, the following code displays a Winform that has a list box populated by the `CustID` column in the `CustStack` stack, and then assigns to the `SelectedCustomer` variable the value that the end user had selected from the list box:

```
FOR ALL NEXT CustID INTO CustStack;  
WINFORM SHOW CustForm;  
SelectedCustomer = CustStack().CustID;
```

`FocIndex` determines which list item is the default choice when the Winform is first displayed at run time. You can control which item is the default by setting the value of `FocIndex`. In the example below, the list box is populated by `CustStack.CustID`, and the default value is taken from the second row of `CustStack.CustID`:

```
FOR ALL NEXT CustID INTO CustStack;  
CustStack.FocIndex = 2;  
WINFORM SHOW CustForm;  
SelectedCustomer = CustStack().CustID;
```

For general information about how an end user would manipulate a list box, see *Using List Boxes* on page 5-4.

Procedure How to Place a List Box in a Winform

1. Select *Listbox*.

You are prompted to point to a location and press the Enter key or PF3 to cancel.

2. Place the cursor at the desired location and press the Enter key or press PF3 to cancel.

If you press the Enter key, the Listbox dialog box is generated:

The screenshot shows a dialog box titled "Listbox". It is divided into two main sections: "Field" and "Stack".

- Field:** A list box containing the following items: CUSTID, CUSTID, LASTNAME, FIRSTNAME, EXPDATE, PHONE, STREET, CITY, STATE. A vertical scrollbar is visible on the right side of this list.
- Stack:** A list box containing the following items: stkcust, stkcust, stkrent.

Below the list boxes, there are several controls:

- A checkbox labeled "[] Size To Fit".
- A "Title:" label followed by a redacted text field.
- An "Object Name:" label followed by a redacted text field.
- Three buttons at the bottom: "Triggers F12", "Ok F4", and "Cancel F3".

Field

The Field combo box lists all the fields in the Master Files that are used by the current procedure. Select the field whose column in the selected stack you want to populate the list box, or type the name of a user-defined column in the selected stack (that is, a column created with the COMPUTE command) that you want to populate the list box.

If you specify a user-defined column, the procedure must declare it before displaying the Winform.

Stack

Select the stack whose column will populate the list box when the Winform is displayed. You can choose from all the stacks that are defined to the Winform.

Size to Fit

Select Size to Fit to size your control to fit the data inside the control.

Title

You can edit the list box's title to another title of your choosing.

Object Name

You may provide a name for the list box, or accept the default name, so that you can refer to the list box within your procedure.

Triggers

You may assign up to twenty-five control-level triggers to each list box. These triggers are invoked when an end user presses the specified function key when the cursor is in the list box. For more information about triggers see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Tip: You may want to avoid assigning a trigger to the Enter key. Because end users must press the Enter key to select an item from a list box, it is recommended that you do not assign a trigger to the Enter key unless you intend end users to trigger the specified function each time they select an item from the list.

Combobox

A combo box is a kind of drop-down list box: it enables the end user to select an item from a list that drops down. A combo box differs from a list box in that a list box is always displayed dropped (that is, fully extended), while in a combo box only the box showing the selected item is displayed until the end user selects the box's v button to drop its list. When the combo box's list is dropped, the end user can select an item from it; the selected item is then displayed in the box at the top of the list, and the list closes.

You can choose to populate the list from:

- A stack column.
- The list of values from a field's ACCEPT attribute in a Master File.

The combo box is populated from the specified source at run time when the Winform is displayed. You identify the source of the combo box's values when you design the Winform.

How you determine which value the end user selected depends on how you populated the combo box. If you populated it using:

- A stack, `FocIndex` points to the stack row that the user selected. For example, the following code displays a Winform that has a combo box populated by the `CustID` column in the `CustStack` stack, and then assigns to the `SelectedCustomer` variable the value that the application user had selected from the combo box:

```
FOR ALL NEXT CustID INTO CustStack;
WINFORM SHOW CustForm;
SelectedCustomer = CustStack().CustID;
```

- An `ACCEPT` attribute, you can use the following syntax to refer to the value that the user selected:

```
formname_comboboxname_ACCEPTS().REC
```

For example, the following code displays a Winform named `CustForm` that has a combo box named `CustListBox`, and then assigns to the `SelectedCustomer` variable the value that the application user had selected from the combo box:

```
WINFORM SHOW CustForm;
SelectedCustomer = CustForm_CustListBox_ACCEPTS().REC;
```

If you populate a combo box from a stack, `FocIndex` determines the default value—that is, the value that will appear in the box when the Winform is first displayed at run time. You can control which value is the default by setting `FocIndex`. In the example below, the combo box is populated by `CustStack.CustID`, and the default value will be taken from the seventh row of `CustStack.CustID`:

```
FOR ALL NEXT CustID INTO CustStack;
CustStack.FocIndex = 7;
WINFORM SHOW CustForm;
SelectedCustomer = CustStack().CustID;
```

For general information about how an end user would manipulate a combo box, see *Using Drop Boxes* on page 5-9.

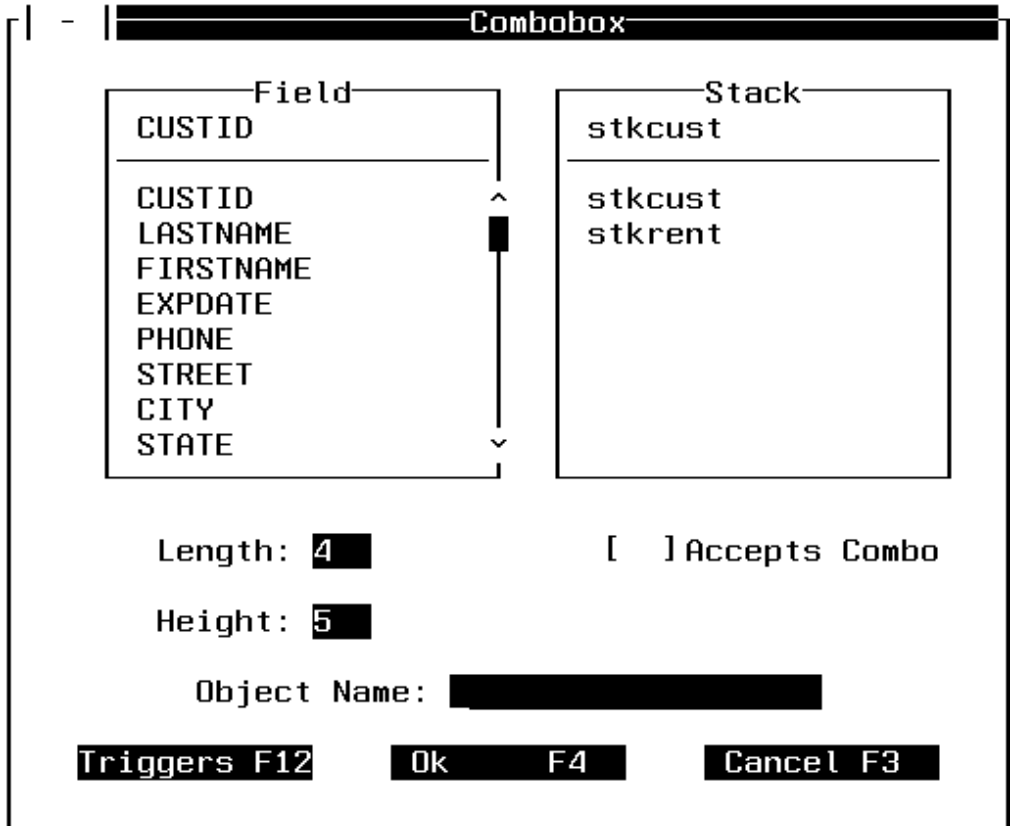
Procedure How to Place a Combo Box in a Winform

1. Select *Combobox*.

You are prompted to point to a location and press the Enter key or PF3 to cancel.

2. Place the cursor at the desired location and press the Enter key or press PF3 to cancel.

If you press the Enter key, the Combobox dialog box is generated:



Field

Field displays all the fields in the Master Files that this procedure uses. Fields that have an ACCEPT attribute in the Master File are displayed with an asterisk (*).

If you are designing this combo box to be populated from:

- An ACCEPT attribute, select the field whose ACCEPT attribute in the Master File you want to populate the combo box.
- A stack, select the field whose column in the selected stack you want to populate the combo box, or type the name of a user-defined column in the selected stack (that is, a column created with the COMPUTE command) that you want to populate the combo box.

If you specify a user-defined column, the procedure must declare it before displaying the Winform.

Stack

Select the stack whose column will populate the combo box. You can choose from all the stacks that are defined to the Winform.

Length

You can edit the combo box's length (that is, the column's width) by placing the cursor at the number following *Length* and entering the new length. For further details concerning column width, see *Length* on page 5-48.

Height

You can edit the combo box's height by placing the cursor at the number following *Height* and entering the new height.

Accepts Combo

If you select Accepts Combo, the combo box will be populated by the list of values in the selected field's ACCEPT attribute in the Master File. This requires that the field's description in the Master File has an ACCEPT attribute that specifies a list of values.

Object Name

You may provide a name for the combo box, or accept the default name, so that you can refer to the combo box within your procedure.

Triggers

You may assign up to twenty-five control-level triggers to each combo box. These triggers are invoked when an end user presses the specified function key when the cursor is in the combo box. For more information about triggers see *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Tip: You may want to avoid assigning a trigger to the Enter key. Because end users must press the Enter key to select an item from a combo box, it is recommended that you do not assign a trigger to the Enter key unless you intend end users to trigger the specified function each time they select an item from the combo box.

Radio Group

A radio button group represents several mutually exclusive options: an end user can select one button from a group.

The radio button group is defined by a stack column:

- Each button's value is provided at run time, when the Winform is displayed, by a stack column. You identify which stack column to use when you design the group.
- The number of buttons in the group is determined at run time, when the Winform is displayed, by the number of rows in the stack.
- The maximum number of buttons that the group can display is determined by the dimensions of the radio button group, the number of columns in which the buttons will be positioned within those dimensions, and the width of the columns, all of which you define when you design the group. Be sure to design the radio button group so that it can accommodate the greatest number of buttons that the application may need to display. If, at run time, the number of rows in the group's source stack exceeds the number of buttons that the group can display, the excess buttons will not be displayed.

When the end user selects a radio button, `FocIndex` points to the stack row that the user selected. For example, the following code displays a Winform that has a radio button group populated by the `CustID` column in the `CustStack` stack, and then assigns to the `SelectedCustomer` variable the value that the end user has selected from the group:

```
FOR ALL NEXT CustID INTO CustStack;  
WINFORM SHOW CustForm;  
SelectedCustomer = CustStack().CustID;
```

FocIndex determines which radio button is selected by default when the Winform is first displayed at run time. You can control which button is the default by setting the value of FocIndex. In the example below, the radio button group is populated by CustStack.CustID, and the default value will be taken from the fifth row of CustStack.CustID:

```
FOR ALL NEXT CustID INTO CustStack;
CustStack.FocIndex = 5;
WINFORM SHOW CustForm;
SelectedCustomer = CustStack().CustID;
```

For general information about how an end user would manipulate a radio button group, see *Using Radio Buttons* on page 5-6.

Procedure How to Place a Radio Button Group in a Winform

1. Select *Radio group*.

You are prompted to point to a location and press the Enter key or PF3 to cancel.

2. Place the cursor at the desired location and press the Enter key or press PF3 to cancel.

If you press the Enter key, the radio button group dialog box is generated:

The screenshot shows a dialog box titled "Radio Group". It contains two main sections: "Field" and "Stack". The "Field" section is a list box containing: CUSTID, CUSTID, LASTNAME, FIRSTNAME, EXPDATE, PHONE, STREET, CITY, STATE. The "Stack" section is a list box containing: stkcust, stkcust, stkrent. To the right of these sections is a "Justification" section with three radio buttons: () Left, (*) Center, () Right. Below the justification section are three controls: "Columns: 1", "Column Width: 10", and "[] Border". At the bottom of the dialog are three buttons: "Triggers F12", "Ok F4", and "Cancel F3". There are also "Title:" and "Object Name:" labels with corresponding input fields.

Field

The Field combo box lists all the fields in the Master Files that are used by the current procedure. Select the field whose column in the selected stack you want to populate the radio button group, or type the name of a user-defined column in the selected stack (that is, a column created with the COMPUTE command) that you want to populate the radio button group.

If you specify a user-defined column, the procedure must declare it before displaying the Winform.

Stack

Select the stack whose column will populate the radio button group when the Winform is displayed. You can choose from all the stacks that are defined to the Winform.

Justification

Justification allows you to choose whether to center, left-justify or right-justify each button's text. Justification is not supported for radio button groups in the current release.

Columns

This determines the number of columns in which the buttons will be lined up.

Column Width

This determines the width of the columns in which the buttons are displayed. For database stack columns, this defaults to the length of the selected field as defined in the Master File plus six, to allow for spacing between columns; for user-defined stack columns it defaults to 13.

If you specify a column width that is less than the field or variable length, any stack values exceeding the specified width will be truncated when they are displayed.

Tip: If you enter a two-digit width, or change the width's units digit, be aware of the cursor position before pressing Enter or a function key. This is because, when typing a units digit in the Column Width field, the cursor's automatic tabbing then moves to the next control in the dialog box, which is the list of fields in the Fields combo box. If you then press the Enter key or a function key to confirm your new width, you will also be resetting the selected field to the field on which the cursor is positioned.

Border

Check Border to put a box around the radio button group.

Title

You can edit the radio button group's title to another title of your choosing.

Object Name

You may provide a name for the radio button group, or accept the default name, so that you can refer to the radio button group within your procedure.

Triggers

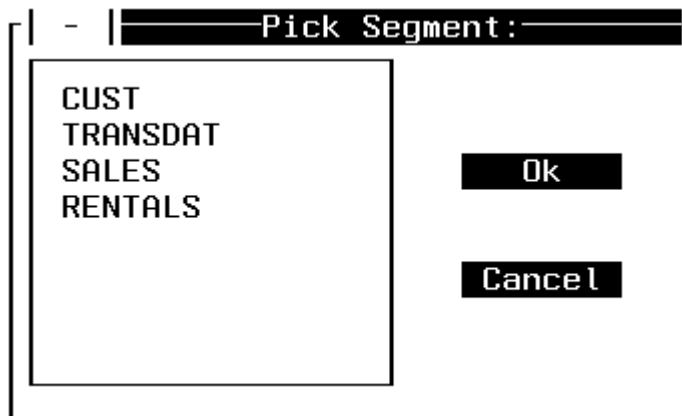
You may assign up to twenty-five control-level triggers to each radio button group. These triggers are invoked when an end user presses the specified function key when the cursor is in the radio button group. For more information about triggers see, *Using Triggers, Button Short Cuts, and System Actions* on page 5-83.

Gen Segment

Select Gen Segment to place all the fields in a segment into a Winform at one time.

Procedure How to Place All the Fields in a Segment in a Winform

Select *Gen Segment*. You are prompted to point to the starting line. Press the Enter key, or press PF3 to cancel. If you press the Enter key, the Pick Segment dialog box is displayed:



Place the cursor under the name of the segment you wish to display and press the Enter key. Press PF3 if you wish to cancel.

For example, if you choose CUST the following is displayed:

```

File  Edit  Forms  Objects  Cases  Help  < MNTUID3 >
-----1-----2-----3-----4-----5-----6-----7-----
                                video
CUSTID      : _____  LASTNAME     : _____
FIRSTNAME   : _____  EXPDATE     : _____
PHONE       : _____  STREET      : _____
CITY        : _____  STATE       : _____
ZIP         : _____

-----1-----2-----3-----4-----5-----6-----7-----
F1-Help  2-AddLn  3-Exit  4-NewFld  5-Move  6-DelObj  7-Up  8-Down  9-DelLn  12-EdtObj
  
```

You may:

- Eliminate any fields you do not wish to display by placing the cursor on these fields and pressing PF6.
- Edit each of the fields to change any options by moving the cursor to the field and pressing PF12. The Change Field dialog box is displayed. For more information about the Change Field dialog box see *Entry Field* on page 5-45.
- Edit the text of the field by moving the cursor under the text and pressing PF12. The Text dialog box is displayed. For more information about the Text dialog box see *Text* on page 5-53.
- Reposition a field by placing the cursor under a field and pressing PF5. You are prompted to point to a new location for the field and press the Enter key or to press PF3 to cancel the transaction. If you point to a new location and press the Enter key, the field moves to the new position on the screen.

Gen Master

Gen Master is similar to Gen Segment except it generates fields for all of the fields in a data source rather than just for a single segment. For example, if you select *Gen Master* and you are working with the VideoTrk data source, a screen similar to the following is displayed:

```

File  Edit  Forms  Objects  Cases  Help
.    1    .    2    .    3    .    genmas    .    5    .    6    .    7    .

CUSTID      :                               LASTNAMEF   :
FIRSTNAME   : _____                     EXPDATE    : _____
PHONE       : _____                     STREET     : _____
CITY        : _____                     STATE      : _____
ZIP         : _____                     TRANSDATE  : _____
PRODCODE    : _____                     TRANSCODE  : _____
QUANTITY    : _____                     TRANSTOT   : _____
MOVIECODE   : _____                     COPY       : _____
RETURNDATE  : _____                     FEE        : _____

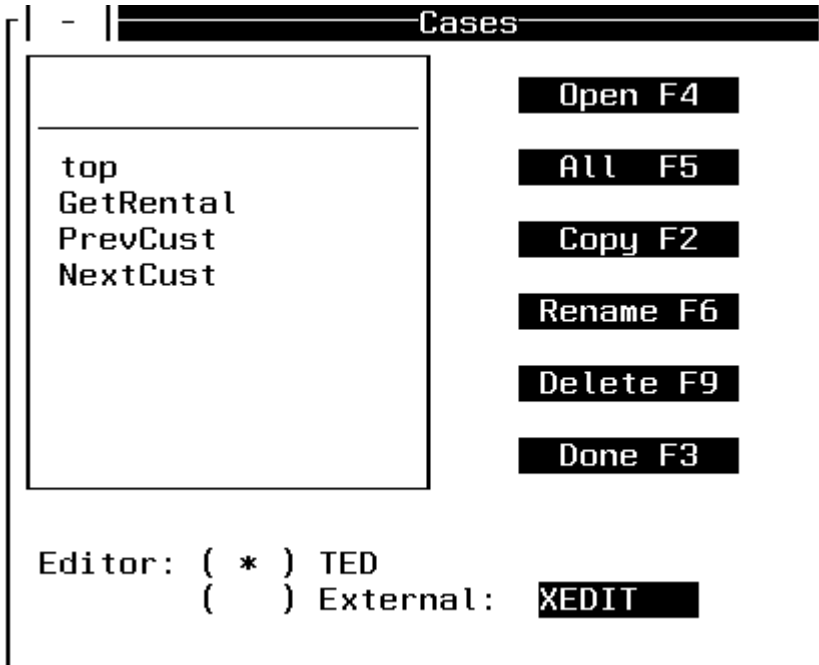
-----1-----2-----3-----4-----5-----6-----7-----
F1=Help 2=AddLn 3=Exit 4=NewFld 5=Move 6=DelObj 7=Up 8=Down 9=DelLn 12=EdtObj

```

You may perform the same actions on these fields that were discussed in *Gen Segment* on page 5-75.

Cases Menu

Selecting the Cases menu enables you to make changes to existing Maintain functions in a Winform file. (Maintain functions—usually just called functions—are known in the Winform Painter as cases.) Selecting the Cases menu yields the Cases dialog box:



Available Cases

The combo box in the left part of the Cases dialog box displays the functions available in the Winform file.

Open

Select Open to open an existing function in order to read or edit it.

To open an existing function, position the cursor under the function you wish to edit and press PF4. This generates the part of the Maintain procedure that contains the specified function. For example, if you were to choose GetRental, the following Winform is displayed:

```

TEMP00  FOCEXEC  A1                               SIZE=10  LINE=0

* * * TOP OF FILE * * *
CASE GetRental                                     -* retrieves data from rental segment
  STACK CLEAR StkRent
  REPOSITION CustID

      -* the following Next retrieves all rental data where Custid in
      -* Videotrck matches the custid displayed on the top of the form
  FOR ALL NEXT CustID TransDate MovieCode INTO StkRent WHERE
    VideoTrk.CustID EQ StkCust(StkCust.FocIndex).CustID
  COMPUTE StkRent.FocIndex = 1; -* puts cursor on first row in grid
ENDCASE
* * * END OF FILE * * *

=====>
TYPING MODE

```

You may now edit the function. When you are finished with your editing, type FILE on the command line and press the Enter key to save the changes. Press PF3 if you do not wish to save the changes.

All

Select All to display the contents of the entire Maintain procedure. In addition to viewing the procedure, you may also make changes to it.

To display the entire Maintain procedure, press PF5. The top section of the procedure is displayed. If the procedure is larger than your screen, you may scroll through it using PF7 and PF8. For example, if you choose *All* using the Cases dialog box at the beginning of this section, the following is displayed:

```
MNTVID1  FOCEXEC  F                               SIZE=155  LINE=0

* * * TOP OF FILE * * *
MAINTAIN FILE VIDEOTRK

    -* this program allows the user to scroll through the customer
    -* history.  No data is updated.

FOR ALL NEXT CustID INTO StkCust    -* retrieves all of the data in the
                                     -* CustID segment and places it in
                                     -* the StkCust stack

PERFORM GetRental

WINFORM SHOW ShowCust              -* displays data to the user

CASE GetRental                      -* retrieves data from rental segment
  STACK CLEAR StkRent
  REPOSITION CustID

    -* the following Next retrieves all rental data where Custid in
    -* Videotrk matches the custid displayed on the top of the form
  FOR ALL NEXT CustID TransDate MovieCode INTO StkRent WHERE
====>

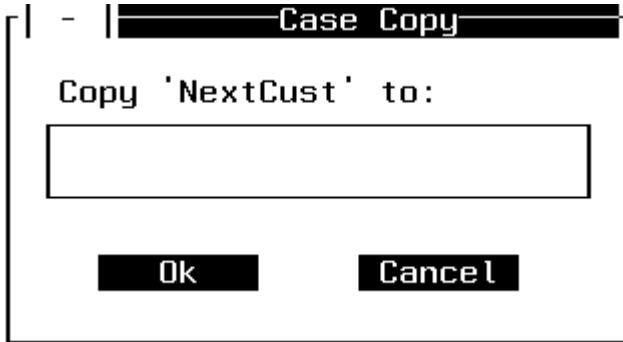
                                     TYPING MODE
```

When you are finished viewing and editing the Maintain procedure, position the cursor at the command line and type the word FILE to save the changes and exit the editor, or press PF3 to cancel the changes you have made and exit.

Copy

Select Copy to copy a function to another function. This can be useful if a function that exists is similar to a function you wish to create. Rather than starting from scratch, you may use the existing function as a basis for the function you wish to create.

To copy a function, place the cursor under the function you wish to copy, and press PF6. Pressing PF6 after positioning the cursor under NextCust yields the Case Copy dialog box:

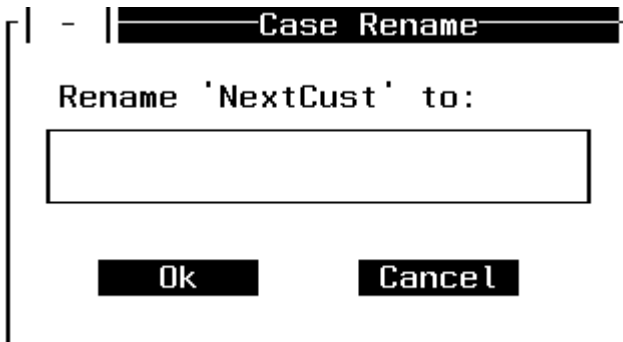


Fill in the name of the function that NextCust is to be copied to. After filling in the name of the function, press PF4 to save it, or PF3 to exit without saving it.

Rename

Select Rename to change the name of a function.

To rename a function, position the cursor under the function name you wish to change, and press PF2. For example, if you choose NextCust, the following Case Rename dialog box is displayed after pressing PF2:

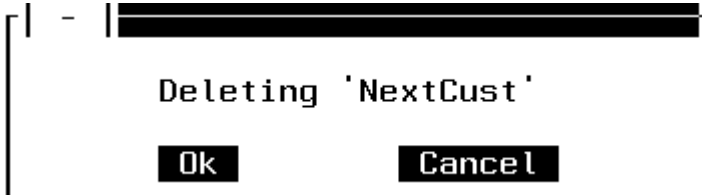


Fill in the new name for NextCust. After filling in the name of the function, press PF4 to save it, or PF3 to exit without saving it.

Delete

Select Delete to delete a function.

For example, if you wish to delete NextCust, the following dialog box is displayed after pressing PF9:



Click *OK* if you wish to delete the function. If you do not wish to, click *Cancel*.

Done

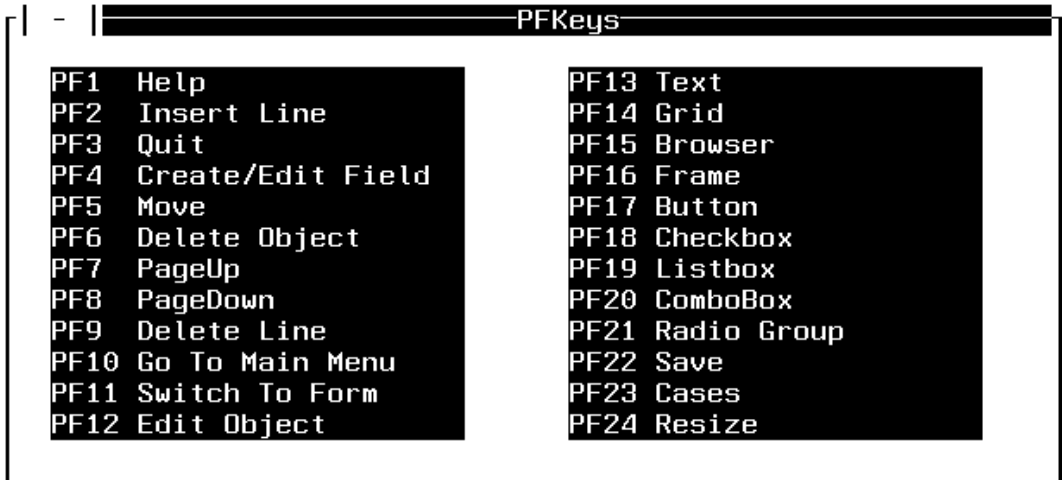
When you have finished, press PF3 to exit from the Cases dialog box.

Editor

You may choose which editor to use for the Maintain procedure. TED is the default. If you wish to use an external editor, fill in the appropriate name and mark the box for *External*. The external editor defaults to your system editor. TED is the only editor available under MVS.

Help Menu

The Help menu displays the settings of the function keys in the Painter. Selecting *Help* yields the PFKeys dialog box:



Using Triggers, Button Short Cuts, and System Actions

Maintain's event-driven processing is based on triggers. Triggers are links between events and actions. An event is something that an end user does, such as pressing a function key; an action is something that the application executes, such as a function. (Functions are called cases by the Winform Painter.)

For example, you might define a trigger that links the PF7 key to the UpdateSalary function. At run time, each time an end user presses PF7, Maintain invokes UpdateSalary. In this case, you have defined pressing the PF7 key as the trigger event, and you have defined the UpdateSalary function as the trigger action. (Trigger actions are also known as event handlers.)

There are several kinds of triggers:

- **Form-level triggers**, which are triggered when an end user presses the specified function key when the cursor is anywhere on the form (on any spot not occupied by a control). You can assign 25 form-level triggers to each form (one trigger each for function keys PF1 through PF24, and one trigger for the Enter key).

- **Control-level triggers**, which are triggered when an end user presses the specified function key when the cursor is on the specified control, such as a particular list box. You can assign 25 control-level triggers to each control (one trigger each for function keys PF1 through PF24, and one trigger for the Enter key). The only exceptions are frames (which do not take any kind of trigger), and buttons (which take button shortcuts).
- **Button shortcuts**. Buttons do not have control-level triggers; instead, you can assign each button one shortcut key. You can assign any function key as a button's shortcut key. Like a control-level trigger, it is assigned to a single control (that is, to the button). Like a form-level trigger, it is active everywhere on the form (on any spot not occupied by a control, other than the button itself).
- **System actions**, which are similar to form-level triggers, but instead of invoking functions, they invoke special system-defined actions that do things like close the current form or exit the current procedure. You can assign 25 system actions to each form (one system action each for function keys PF1 through PF24, and one system action for the Enter key).

It is possible to assign the same key to several triggers and a system action. What would happen if an end user pressed that key? What action(s) would the key execute? That would be determined by the general order of precedence for resolving a key's assignments: control-level trigger, then form-level trigger or button shortcut, then system action. When an end user presses a function key, if:

1. The cursor is on a control, and one of the control's triggers has been assigned to that key, that trigger's Maintain function is performed. (Maintain functions—usually simply called functions—are referred to in the Winform Painter as cases.)

If a system action has also been assigned to that key, the system action is performed immediately following the trigger's function. (If the trigger function closes the Winform, the system action is not performed.)

2. The cursor is not on a control—or it is on a control for which no trigger has been assigned to that key—but a form-level trigger or a button shortcut has been assigned to that key, the form-level trigger or button trigger's function is performed.

If a system action has also been assigned to that key, the system action is performed immediately following the trigger's function. (If the trigger function closes the Winform, the system action is not performed.)

3. No form-level triggers have been assigned to that key, and no control-level triggers for that key are in effect, but a system action *has* been assigned to that key, the system action is performed.

Specifying Triggers

Selecting Triggers in the Form menu, or clicking the Triggers button or pressing PF12 when creating or editing a control (other than a button) displays the Triggers dialog box:

Triggers					
Key	System Action	Trigger	Key	System Action	Trigger
Enter		getrental			
PF1	accept	prevcust	PF13		
PF2		nextcust	PF14		
PF3	exit		PF15		
PF4	return		PF16		
PF5			PF17		
PF6			PF18		
PF7	backward		PF19		
PF8	forward		PF20		
PF9			PF21		
PF10	left		PF22	fieldleft	
PF11	right		PF23	fieldright	
PF12			PF24	ontop	

Ok F4 Cases F5 Cancel F3

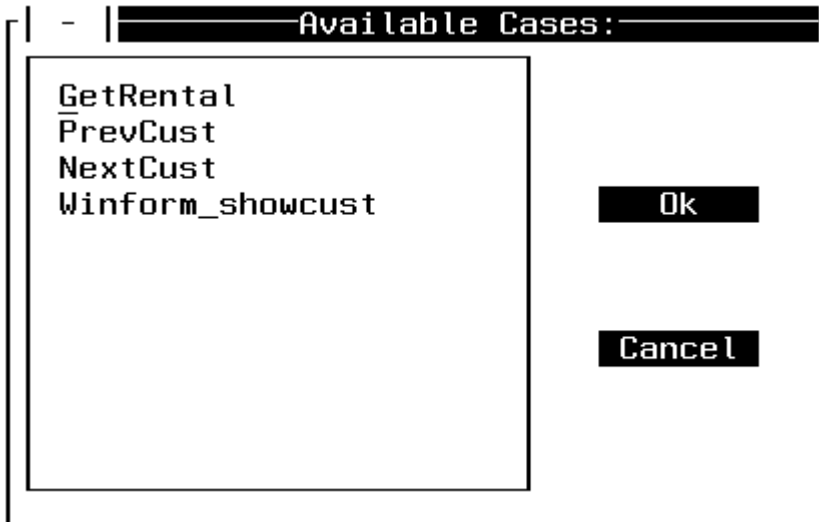
-----+-----1-----+-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----7-----+-----

Enter case names against the corresponding keys

All of the function keys are displayed with their corresponding Maintain functions. (Maintain functions—often simply called functions—are referred to in the Winform Painter as cases.) You may change, add, or delete functions. When you are done, press PF4. Press PF3 to cancel the transaction.

For reference, you can also see the system actions assigned to each key.

To view the available functions, place the cursor in the shaded Trigger column, on the row of the desired PF key, and press PF5. This displays the Available Cases dialog box:



Select a function from the list and press PF4. This assigns the function to the PF key you specified in the Triggers dialog box. Use PF8 to scroll forwards and PF7 to scroll backwards in the Available Cases dialog box. Press PF3 to cancel the action and return to the Triggers dialog box.

You can easily view, edit, delete, and rename cases using the Cases menu. For more information, see *Cases Menu* on page 5-78.

Specifying System Actions

Selecting Actions in the Forms menu displays the System Actions dialog box:

System Actions					
Key	Description	Action	Key	Description	Action
Enter					
PF1	Accept List	accept	PF13		
PF2			PF14		
PF3	Exit Focexec	exit	PF15		
PF4	Close Winform	return	PF16		
PF5			PF17		
PF6			PF18		
PF7	Scroll Grid Up	backward	PF19		
PF8	Scroll Grid Down	forward	PF20		
PF9			PF21		
PF10	Scroll Grid Left	left	PF22	Scroll Field Left	fieldleft
PF11	Scroll Grid Right	right	PF23	Scroll Field Right	fieldright
PF12			PF24	Move Form to Top	ontop

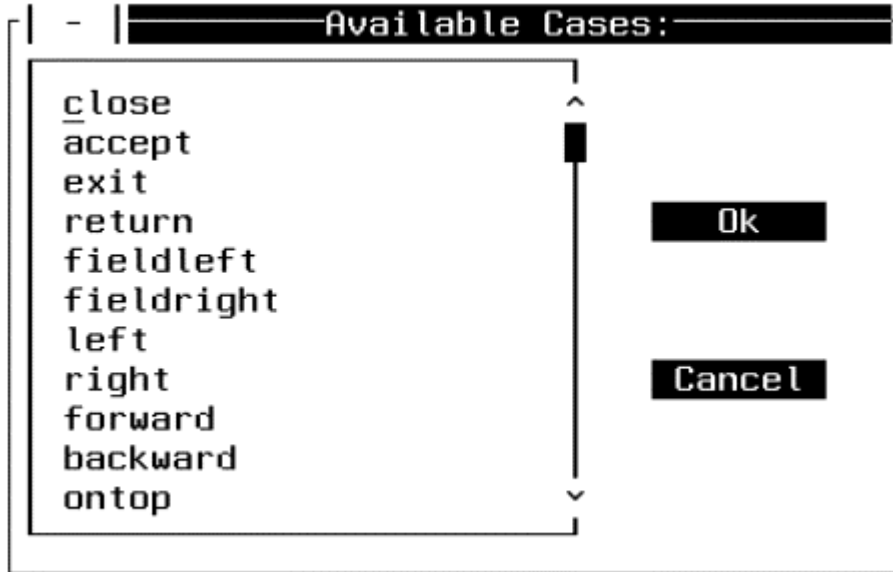
Ok F4 Actions F5 Cancel F3

-----1-----2-----3-----4-----5-----6-----7-----

Enter System Actions against the corresponding keys

All of the function keys are displayed with their corresponding system actions. Some system actions are assigned by default. You may change, add, or delete system actions. When you are done, press PF4. Press PF3 to cancel the transaction.

To view the available system actions, place the cursor in the shaded Action column, on the row of the desired PF key, and press PF5. This displays the Available Cases dialog box, which lists all system actions:



Select a system action from the list and press PF4. This assigns the system action to the PF key you specified in the Systems Actions dialog box. Use PF8 to scroll forward and PF7 to scroll backward in the Available Cases dialog box. Press PF3 if you want to close it and return to the Systems Actions dialog box without assigning a system action.

Reference Available System Actions

You can assign the following system actions to system action keys:

System action	What it does
Close	Closes the currently active Winform and returns control to the function that opened it.
Accept	When the cursor is on a field for which Accept data validation information is available, displays a list of values from which the end user can select. For more information about making Accept data validation information available for a field, see <i>Accepts</i> on page 5-49.
Exit	Terminates the current Maintain procedure; if this procedure was called by another procedure, it returns control to the calling procedure.
Return	Is a synonym for the Close system action.
FieldLeft	When the cursor is on a field or grid cell in which the data is longer than the width of the field or cell, scrolls the data to the left.
FieldRight	When the cursor is on a field or grid cell in which the data is longer than the width of the field or cell, scrolls the data to the right.
Left	When the cursor is on a grid, scrolls the grid one column to the left.
Right	When the cursor is on a grid, scrolls the grid one column to the right.
Forward	When the cursor is on a grid, scrolls down the grid's columns the full height of the grid.
Backward	When the cursor is on a grid, scrolls up the grid's columns the full height of the grid.
OnTop	If other Winforms are displayed on top of the current Winform, moves the current Winform to the top of the pile.
Close_Winform	Is a synonym for the Close system action.

Example Trigger and System Action Precedence

Imagine that, in a new application, you assign the UpdateEmployee function to the form-level PF5 trigger. At run time, if the end user presses PF5 while the cursor is on the Winform background, Maintain will invoke UpdateEmployee.

Imagine that you also assign the ValidateCheckBox function to a check box's PF5 trigger. Furthermore, you assign the Close action to the PF5 system action. At run time, if the end user presses PF5 while the cursor is on the:

- **Check box**, Maintain will invoke the ValidateCheckBox function. After control returns from ValidateCheckBox, Maintain will execute the Close system action, which closes the Winform.
- **Winform background**, Maintain will invoke the UpdateEmployee function. After control returns from UpdateEmployee, Maintain will execute the Close system action, which closes the Winform.

CHAPTER 6

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 standards, including:

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

Master Files are not part of the Maintain language and are subject to different rules. The language rules for Master Files are discussed in the *Describing Data* manual.

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 uppercase and lowercase.

This manual uses mixed case for user-defined names, in order to illustrate the use of mixed case; however, the Winform Painter displays some names in uppercase. The same item may be shown in mixed case in the manual and uppercase on the screen; the difference is insignificant.

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.'

Note: While you don't need to worry about case in Maintain procedures, any Master Files that Maintain accesses must be in uppercase.

Specifying Names

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

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

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

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

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

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

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

Maintain requires that the qualification character be a period (.); the QUALCHAR parameter of the SET command must therefore be set (to the default).

If a qualified name cannot fit onto the current line, you can break the name at the end of any one of its components, and continue it onto the next line. The continued name must begin with the qualification character. In the following example, the continued line is indented for 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 Winform in which it is found. For example, if a button named UpdateButton is in a form named CreditForm, you could refer to the button as:

```
CreditForm.UpdateButton
```

- **Member functions and member variables.** When referring to 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 Winform *objectname.functionname* or *objectname.variablename*).
- **Fields and columns.** You can qualify a variable name with the name of the data source, segment, and/or stack in which it is found, using a period (.) as the qualification character.

Qualification is important when you are working with two data sources in one Maintain procedure, and the data sources have field names in common; 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 Winforms in the same procedure include identically-named controls.

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

```
NEXT Employee.JobCode;
```

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

```
stackname(row) . datasourcename . segmentname . fieldname
```

If you refer to a field using a single qualifier (such as Sales) in the example

```
Sales.Quantity
```

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

- **Truncated names.** You must spell 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 Winforms.

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

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

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	OBJECT	OF	OMITS	ON
OR	PERFORM	QUIT	REPEAT	REPOSITION

What Can You Include in a Procedure?

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 7, *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 6-8.
- **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 Dialogue Manager commands preceding the MAINTAIN command. Dialogue Manager commands are described in the *Developing Applications* manual.

Multi-line Commands

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

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

In the following example, all continued lines are indented for 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, supplying it is recommended.

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

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

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

- **An end keyword.** Some commands, such as BEGIN, CASE, and REPEAT, bracket a block of code. You indicate the end of the block by supplying the 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 7, *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 7

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.

Language Summary

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

Defining a Procedure

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

MAINTAIN

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

END

Terminates the execution of a Maintain procedure.

Defining a Maintain Function (a Case)

The following command defines Maintain functions:

CASE

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

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. (Functions are also known as cases.)

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.

The CALL command is advantageous because common code can be shared by many developers, which speeds up both development and maintenance time. For example, a generalized error message display procedure could be used by all Maintain developers. After passing a message to the generalized procedure, the procedure would handle message display. The developers do not need to worry about how to display the message, and the error messages always look consistent to the end users. A second advantage to a modular design is that code that is not executed very frequently can be placed in separate programs in order to reduce the size of the system. Reducing the size makes maintenance easier because the code is easier to understand. Furthermore, if the code is not executed, less memory is used.

Executing Procedures

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

CALL

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

COMPILE/MNTCON COMPILE

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

The difference between COMPILE and MNTCON COMPILE is that COMPILE only works on procedures with file type or ddname FOCEXEC; whereas MNTCON COMPILE works on procedures with file type or ddname FOCEXEC or MAINTAIN.

EX/MNTCON EX

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

The difference between EX and MNTCON EX is that EX only works on procedures with file type or ddname FOCEXEC, whereas MNTCON EX works on procedures with file type or ddname FOCEXEC or MAINTAIN.

RECOMPILE

Recompiles a Maintain procedure that had been compiled under an earlier release of Maintain, in order to improve performance. This command is outside the Maintain language, but is described here for your convenience.

RUN/MNTCON RUN

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

The difference between RUN and MNTCON RUN is that RUN works on compiled Maintain and FOCUS procedures, whereas MNTCON RUN only works on compiled Maintain procedures.

Encrypting Files

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

ENCRYPT

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.

Winforms

The following command is responsible for presentation logic:

WINFORM

Displays a form by which end users can read, enter, and edit data, and manipulates control properties. (Controls are also known as Winform objects.)

Defining Classes

The following command enables you to define classes:

DESCRIBE

Defines classes and data type synonyms.

Creating Variables

The following commands enable you to create variables:

DECLARE

Creates local and global variables, including objects.

COMPUTE

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

Assigning Values

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

`COMPUTE`

Assigns values to existing variables.

Manipulating Stacks

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

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

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

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

copies 100 records of the Indata stack starting from the 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 change position in the data source.

The commands are:

`NEXT`

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

`MATCH`

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

`REPOSITION`

Changes 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.PRE_MATCH`

Turns off preliminary database operation checking before an update.

`SYS_MGR.GET_PRE_MATCH`

Determines whether prematch checking is on or off.

`SYS_MGR.ENGINE`

Passes SQL commands directly to a DBMS.

`SYS_MGR.DBMS_ERRORCODE`

Retrieves a DBMS return code after an operation.

Conditional Actions

The conditional commands are:

IF

Issues a command depending on how an expression is evaluated.

ON MATCH

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

ON NOMATCH

Defines the action to take if the prior MATCH fails.

ON NEXT

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

ON NONEXT

Defines the action to take if the prior NEXT fails.

Writing Transactions

The commands that can be used to control transactions are:

INCLUDE

Adds one or more new data source records.

UPDATE

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

REVISE

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

DELETE

Deletes one or more records from the data source.

COMMIT

Makes all data source changes since the last COMMIT permanent.

ROLLBACK

Cancels all data source changes made since the last COMMIT.

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.PRE_MATCH`

Turns off preliminary database operation checking before an update.

`SYS_MGR.GET_PRE_MATCH`

Determines whether prematch checking is on or off.

`SYS_MGR.ENGINE`

Pass SQL commands directly to a DBMS.

`SYS_MGR.DBMS_ERRORCODE`

Retrieves a DBMS return code after an operation.

Changing the Environment

You can change the Maintain environment using the following command:

`SYS_MGR.FOCSET`

Sets FOCUS parameters.

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.

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

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

Example Nested BEGIN Blocks

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

```
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 using variables: local variables that you pass between them and the global transaction variables FocError, FocErrorRow, and FocCurrent. CALL allows you to link modular procedures, so each procedure can perform its own set of discrete operations within the context of your application.

For additional information about requirements for passing variables, see *Executing Other Maintain Procedures* in Chapter 2, *Maintain Concepts*.

Syntax

CALL Command

The syntax of the CALL command is

```
CALL procedure [FROM var_list] [INTO var_list] [;]
    var_list: {variable} [{variable} ... ]
```

where:

procedure

Is the name of the Maintain procedure to run.

FROM

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

INTO

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

var_list

Are the scalar variables and stacks, 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.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 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 using its parameters, and can pass values from a Maintain function using its parameters and its return value.

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

- Issuing a PERFORM or GOTO command.
- Calling the function directly.
- Calling the function as a trigger action or event handler.

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 a trigger), or called directly, control returns to the point immediately following the PERFORM, trigger, or function reference.
- **Branch.** If the function was called by a simple branch command (that is, by a GOTO command), 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 a trigger action, it cannot take parameters.

RETURNS result/t

Specifies that the function returns a value. *result* is the name of the variable being returned, and *t* is the 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 a trigger action, it cannot return a value.

declarations

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

commands

Is one or more commands, except for CASE, DESCRIBE, END, MAINTAIN, and MODULE.

;

Terminates the 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 *Terminating a Command's Syntax* in Chapter 6, *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 reaches the end of the function, it returns to the command following PERFORM.
- **GOTO** transfers control to another function or to the end of the current function. Unlike the PERFORM command, it does not return the control of the command that called the function.

Calling a Function: Flow of Control

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

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

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

Passing Parameters to a Function

In general, 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.

Note: There is one exception to this behavior. If you declare a function parameter (in the Function Editor or a CASE command) with one data type, but at run time you pass the function a value of a different data type, the 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, An 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 using 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 run a Maintain procedure, the procedure begins by executing its Top function. Every Maintain procedure has a Top function. Top does not take or return parameters. You can choose to define the Top function:

- **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.

For example:

```

CASE Top
.
.
.
ENDCASE

```

- **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 non-executable 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 6, *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 DB2 data source Salary, the success or failure of the COMMIT for Salary is independent of the success of the COMMIT for Employee and JobFile. This is known as a broadcast commit.
- COMMIT is automatically issued when a procedure does not contain any COMMIT commands, and the application is exited normally. This means an error did not cause program termination. If a procedure does not contain any COMMIT commands and it is terminated abnormally (for example if the system has run out of memory), a COMMIT is not issued. When a called procedure is exited, an automatic COMMIT is not issued. COMMIT is only issued when exiting the application.
- The variable FocCurrent is set after a COMMIT finishes. If the COMMIT is successful, FocCurrent is set to zero. If FocCurrent is not zero, the COMMIT failed, and all of the records in the logical unit of work will be rolled back because an internal ROLLBACK will be issued.

COMPILE

The COMPILE command creates a compiled procedure with a file type of FOCCOMP (under CMS) or allocated to ddname FOCCOMP (under OS/390). You can reduce the time needed to start a procedure that contains Winforms by compiling the procedure. The more frequently the procedure will be run, the more time you save by compiling it.

Because COMPILE only compiles Maintain procedures with a file type or ddname of FOCEXEC, we recommend that you use the MNTCON COMPILE command. See *MNTCON COMPILE* on page 7-61.

This command is outside the Maintain language, but is described here for your convenience. You cannot issue this command 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

- **MNTCON COMPILE** also compiles a Maintain procedure.
- **RECOMPILE** enables you to recompile a procedure to run with maximum efficiency under a later release of Maintain.
- **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 7-33 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. It must resolve to a single field or stack cell. If the variable does not have a prefix, it is assumed to be a Current Area variable. Even when a prefix is provided, it is the corresponding column in the Current Area that receives the value. The only two variables that can be assigned a value are a stack cell and a Current Area column. 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 built-in formats and edit options (except for TX) as described for the Master File FORMAT attribute in *Describing Data*; 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.

Specifying DFC *cc* is optional if the data type is a built-in format (such as alphanumeric or integer). 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 *Developing 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*.

Specifying YRT *yy* is optional if the data type is a built-in format (such as alphanumeric or integer). 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 *Developing Applications*.

missing

Is used to allow or disallow null values. This is optional if the data type is a built-in format (such as alphanumeric or integer). 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 alphanumeric and date and time variables. If it is included, its default value is null.

ON

Sets the default value to null.

COMPUTE

OFF

Sets the default value to zero or a space.

NEEDS

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

SOME

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

ALL

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

DATA

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

=

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 8, *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. For more information, see *Describing Data*.
- **DECLARE** creates local and global variables.

Using COMPUTE to Call Functions

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

```
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 6, *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 (FROM) stack into the target (INTO) stack. If you omit both FOR and STACK, only the first row of the source stack is copied.

int

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

ALL

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

STACK

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

FROM

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

INTO

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

stk

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

row

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

CURRENT

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

WHERE

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

expression

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

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 (FROM) stack is copied because this is the default when a prefix is not specified for the COPY command. The data is copied into the first row of the Temp stack because the first row is the default when a row number is not supplied for the target (INTO) stack:

```
COPY FROM Emp INTO Temp;
```

Example Copying a Row Into the Current Area

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

```
COPY FROM Emp(10) INTO CURRENT;
```

Example Copying Rows Based on Selection Criteria

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

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

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

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

Example Appending One Stack to Another

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

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

Reference Usage Notes for COPY

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

COPY

- If the source (FROM) stack is the Current Area, the only Current Area fields that are copied are those that have a corresponding column name in the target (INTO) stack.
- If the target (INTO) stack is not subscripted, the data is copied into the first row in the stack. If the target (INTO) stack is subscripted, the copied row or rows are inserted at this row.
- If the COPY command specifies the command output destination as a row or rows of an existing stack that already have data in them, then the old data in these rows is overwritten with the new data when the COPY is executed.
- If the source (FROM) stack has fewer columns than the target (INTO) stack, the columns that do not have any data are initialized to blank, zero or null (missing) as appropriate.
- Source (FROM) stack rows will overwrite the specified target (INTO) stack rows if they already exist.
- If the COPY command creates rows in the target (INTO) stack, and the target (INTO) stack contains columns that are not in the source (FROM) stack, those columns in the new rows will be initialized to their default values of blank, zero, or null (missing).
- If the source (FROM) stack has more columns than the target (INTO) stack, only corresponding columns are copied.
- The FOR prefix copies rows from the source (FROM) stack one row at a time, not all at the same time. For example, the following command

```
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 7-33 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; see *Specifying Names* in Chapter 6, *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 alphanumeric 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 *Developing Applications*.

DECLARE

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 *Developing 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 alphanumeric and date and time variables. If it is included, its default value is null.

ON

Sets the default value to null.

OFF

Sets the default value to zero or a space.

NEEDS

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

SOME

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

ALL

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

DATA

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

()

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

Reference **Commands Related to DECLARE**

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

Local and Global Declarations

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

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

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

- **Global** (that is, known to all the functions in the procedure). To declare a global variable, place the DECLARE command outside of a function (for example, at the beginning of the procedure prior to all functions), or define it using the COMPUTE command anywhere in the procedure.

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

DELETE

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

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

row

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

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 Winform

The first example prompts the user for the employee ID in the EmployeeIDForm form. If the employee is already in the data source, all records for that employee are deleted from the data source. This includes the employee'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 from a form. Control is then passed back to EmployeeIDForm.

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

```
MAINTAIN 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 there is a concurrency conflict at COMMIT time, the transaction variable FocCurrent is set to a non-zero value.
- A DELETE command cannot have more than one input (FROM) stack.
- When a DELETE command is complete, the variable FocError is set. If the DELETE is successful (the records to be deleted exist in the data source) then FocError is set to zero. If the records do not exist, FocError is set to a non-zero value. If the DELETE operation was set-based, Maintain sets FocErrorRow to the number of the row that failed.
- Maintain requires that data sources to which it writes have unique keys.

Reference Commands Related to DELETE

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

DESCRIBE

The DESCRIBE command enables you to define 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). (Functions are also known as cases.)

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; see *Specifying Names* in Chapter 6, *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; see *Specifying Names* in Chapter 6, *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 7-15 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; see *Specifying Names* in Chapter 6, *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 ENDD DESCRIBE command is required.

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

ENDD DESCRIBE

Ends the class definition if it includes member functions. If it omits member functions, the ENDD DESCRIBE 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;
```

END

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.
- **COMPUTE** creates global variables, including global objects, and assigns values to existing variables.

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.
- **CALL** is used to call one procedure from another.

EX

You can run an uncompiled Maintain procedure with the FOCEXEC file type by issuing the EX command (which is sometimes lengthened to EXEC).

Because EX only executes Maintain procedures with a file type or ddname of FOCEXEC, we recommend that you use the MNTCON EX command. See *MNTCON EX* on page 7-62.

This command is outside the Maintain language, but is described here for your convenience. You cannot issue this command from within a Maintain procedure.

Syntax **EX Command**

The syntax of the EX command is

```
EX[EC] procedure_name
```

where:

EC

Is an optional extension of the command, and is provided only for user readability.

procedure_name

Is the name of the Maintain procedure. The Maintain procedure must have the file type FOCEXEC.

Reference **Commands Related to EX**

- **COMPILE** compiles Maintain procedures with the FOCEXEC file type.
- **MNTCON EX** executes Maintain procedures with the MAINTAIN file type.

Example **Using EX**

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

```
EX EmpInfo
```

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

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

```
FOR ALL NEXT Emp_ID Curr_Sal INTO Pay;  
COMPUTE Pay.NewSal/D12.2=;  
REPEAT Pay.FocCount Cnt/I4=1;  
    COMPUTE Pay(Cnt).NewSal = 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 a global variable: you do not need to pass it between procedures.

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 a global variable; you do not need to pass it between procedures.

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 a global variable; you do not need to pass it between procedures.

FocIndex

The FocIndex stack variable is a pointer to the current instance in a stack. 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.

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 and PERFORM* on page 7-46 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 7-15.

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 7-90.

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 7-90.

;

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 7-20.

Reference Commands Related to GOTO

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

Using GOTO With Data Source Commands

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

```
ON MATCH GOTO MatchEdit;
```

GOTO and ENDCASE

When control is transferred to a function with the GOTO command, every condition for exiting that function must contain a command indicating where control should be passed to. If an ENDCASE command is reached by either GOTO or normal program flow, and Maintain has not received any instructions as to where to go next, Maintain takes a default action and exits the procedure. ENDCASE is treated differently when GOTO and PERFORM are combined. See *PERFORM* on page 7-87 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 from control point immediately following it. The scope includes any additional PERFORM commands nested within it.

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

If, when you code your application, you cannot know every potential 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 8, *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 8, *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";
```

This example can be executed 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.

path_spec

Identifies the path to be added to the data source. To identify a path, specify its anchor and target segments. (You cannot specify a unique segment as the anchor.) If the path contains only one segment, and the anchor and target are identical, simply specify the segment once. (For paths with multiple segments, if you wish to make the source code clearer to readers, you can also specify segments between the anchor and target.)

To add a unique segment instance to a data source, you must explicitly specify the segment in *path_spec*. Otherwise, the unique segment instance will not be added even if it is on the path between the anchor and target segments. This preserves the advantage of assigning space for a unique segment instance only when the instance is needed.

To specify a segment, provide the name of the segment or of a field within the segment.

FROM

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

stack

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

row

Is a subscript that specifies the first stack row to add to the data source.

INCLUDE

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 instructs the system to start in the second row of the stack; the `FOR 3` instructs the system 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 is input without a prior confirmation that it does not already exist in the data source, one or more of the rows in the stack may be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. For all of the rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work and `ROLLBACK` issued if the entire set was not accepted. After an `INCLUDE`, the transaction variable `FocError` is given a value. If the `INCLUDE` is successful, `FocError` is zero. If the `INCLUDE` fails (for example, if the key values already exist in the data source), `Maintain` assigns a non-zero value to `FocError`, and—if the include was set-based—assigns the value of the row that failed to `FocErrorRow`. If at `COMMIT` time there is a concurrency conflict, `Maintain` sets `FocCurrent` to a non-zero value.

Example Adding Multiple Segments

This example shows how data is added from two segments in the same path. The data comes from a stack named `EmpInfo` and the entire stack is used. When the `INCLUDE` is complete, the variable `FocError` is checked to see if the `INCLUDE` was successful. If it failed, a general error handling function is called:

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

Example Adding Data From the Current Area

The user is prompted for the employee'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.
- When an INCLUDE command is complete, the variable FocError is set. If the INCLUDE is successful (the records to be added do not exist in the data source) then FocError is set to zero. If the records do exist, FocError is set to a non-zero value, and—if it is a set-based INCLUDE—FocErrorRow is set to the number of the row that failed.
- Maintain requires that data sources to which it writes have unique keys.

Reference Commands Related to INCLUDE

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

Data Source Position

A Maintain procedure always has a position either within a segment or just prior to the first segment instance. If data has been retrieved, the position is the last record successfully retrieved on that segment. If a retrieval operation fails, the data source position remains unchanged.

If an INCLUDE is successful, the data source position is changed to the new record. On the other hand, if the INCLUDE fails, it might be because there is already a record in the data source with the same keys. In this case the attempted retrieval prior to the INCLUDE is successful, and the position is on that record. Therefore the position in the data source changes.

Null Data

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 alphanumeric and date and time fields) or zero (for numeric fields).

INFER

Stacks are array variables containing rows and columns. When defining a stack and its structure, provide a name for the stack and a name, format, and order for each of the columns in the stack. Stacks can be defined in two ways:

- Performing actual data retrieval with the NEXT command, the stack is defined and populated at the same time. The stack is defined with all the segments that are retrieved. This is convenient when the procedure is processing on the same physical platform as the data source.
- If the procedure referring to a stack does not retrieve data, you must issue the INFER command to define the 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, and must 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, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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.
- **COPY** can be used to copy data from one stack to another.
- **COMPUTE** can be used to define the contents of a stack for non-data source fields.

Defining Non-Data Source Columns

To define stack columns in a procedure for non-data source fields—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.

'
Is used to separate Master File names.

varlist

Is the variables, both Current Area 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 must 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.
- **CALL** is used to call one procedure from another.

Specifying Data Sources With the MAINTAIN Command

The MAINTAIN command does not require that any parameters are supplied; 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 is complete, control returns to the item that follows the CALL command in the calling procedure.

For additional information about calling one procedure from another, see *Executing Other Maintain Procedures* in Chapter 2, *Maintain Concepts*, and for information about the CALL command, see *CALL* on page 7-12.

Example Passing Variables Between Procedures

You can pass stacks and variables between procedures by using FROM and INTO variable lists. In the following example, when the CALL Validate command is reached, control is passed to the procedure named Validate along with the Emps stack. Once Validate is complete, the data in the stack ValidEmps is sent back to the calling procedure. Notice that the calling and called procedures both have the same FROM and INTO stack names. Although this is not required, it is good practice to avoid giving the same stacks different names in different procedures.

The calling procedure contains:

```

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.

MATCH

stack

Is a stack name. Only one stack can be specified for each FROM or INTO phrase. The stack name should have a subscript specifying which row is to be used. If a stack is not specified, the values retrieved by the MATCH go into the Current Area.

row

Is a subscript that specifies which row is used. The first row in the stack is matched against the data source if the FROM stack does not have a subscript. The data is placed in the first row in the stack if the INTO stack does not have a subscript.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 the value for Pay_Date from the Pay_Date field, which is in the Current Area. After the match is found, all of the field values in the PayInfo segment are copied from the data source into the Current Area:

```
MATCH Pay_Date;
```

The next example shows a MATCH on the key in the EmplInfo segment. It gets the value for Emp_ID from the Emp_ID column in the Cnt row of the Stackemp stack. After the match is found, all of the fields in the 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.
- **REPOSITION** changes data source position to be at the beginning of the chain.

How the MATCH Command Works

When a MATCH command is issued, Maintain tries to retrieve a corresponding record from the data source. If there is no corresponding value and an ON NOMATCH command follows, the command is executed.

The MATCH command looks through the entire segment to find a match. The NEXT command with a WHERE qualifier also locates a data source record, but does it as a forward search. That is to say, it starts at its current position and moves forward. It is not an exhaustive search unless positioned at the start of a segment. This can always be done with the REPOSITION command. A MATCH is equivalent to a REPOSITION on the segment followed by a NEXT command with a WHERE phrase specifying the key. If any type of test other than the equality test that the MATCH command provides is needed, the NEXT command should be used.

MNTCON COMPILE

The MNTCON COMPILE command creates a compiled Maintain procedure which, under:

- **OS/390** is allocated to ddname FOCCOMP.
- **CMS** has a file type of FOCCOMP.

You can reduce the time needed to start a Maintain procedure that contains forms by compiling the procedure. The more frequently the Maintain procedure will be run, the more time you save by compiling it.

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

Syntax MNTCON COMPILE Command

The syntax of the MNTCON COMPILE command is

```
MNTCON COMPILE procname
```

where:

procname

Is the name of a Maintain procedure. First, the MNTCON COMPILE command looks for a Maintain procedure with a MAINTAIN file type or ddname. If it doesn't find one, it looks for a Maintain procedure with a FOCEXEC file type or ddname.

Reference Commands Related to MNTCON COMPILE

- **MNTCON RUN** executes compiled Maintain procedures.
- **MNTCON EX** executes uncompiled Maintain procedures.

MNTCON EX

You use the MNTCON EX command to run an uncompiled Maintain procedure.

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

Syntax

MNTCON EX Command

To run an uncompiled Maintain procedure (with a MAINTAIN or FOCCOMP file type or ddname), use the following syntax

```
MNTCON EX procname
```

where:

procname

Is the name of a Maintain procedure. First, the MNTCON EX command looks for a Maintain procedure with a MAINTAIN file type or ddname. If it doesn't find one, it looks for a Maintain procedure with a FOCEXEC file type or ddname.

Reference **Commands Related to MNTCON EX**

- **MNTCON COMPILE** compiles Maintain procedures.
- **MNTCON RUN** executes compiled Maintain procedures.

MNTCON RUN

You use the MNTCON RUN command to run a Maintain procedure that has been compiled.

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

Syntax MNTCON RUN Command

To run a Maintain procedure that has been compiled (with a FOCCOMP file type or ddname), use the following syntax

```
MNTCON RUN procname
```

where:

procname

Is the name of a Maintain procedure.

Reference Commands Related to MNTCON RUN

- **MNTCON COMPILE** executes compiled Maintain procedures.
- **MNTCON EX** executes uncompiled Maintain procedures.

MODULE

The MODULE command accesses a source code library so 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 library is a FOCEXEC file, and its naming and allocation requirements are the same as those for FOCEXEC files generally.

If a library is specified multiple times in a MODULE command, Maintain will include the library only once in order to avoid a loop.

Reference **Commands Related to MODULE**

- **DESCRIBE** defines classes; you can use DESCRIBE to include classes in a library.
- **CASE** defines a function; you can use CASE to include functions in a library.

What You Can and Cannot Include in a Library

You can include most Maintain language commands and structures in a library. However, there are some special opportunities and restrictions of which you should take note:

- **Other libraries.** You can place one library within another, and can nest libraries to any depth. For example, to nest library B within library A, issue a MODULE IMPORT B command within library A.
If a given library is specified more than once in a series of nested libraries, Maintain will only include the library once in order to avoid a loop.
- **Top function.** Because a library is a nonexecutable procedure, it has no Top function.
- **Forms.** A library cannot contain forms.
- **Data sources.** A library cannot refer to data sources. For example, it cannot contain data source commands (such as NEXT and INCLUDE) and cannot refer to data source stacks.

NEXT

The NEXT command selects and reads segment instances from a data source. You can use NEXT to read an entire set of records at a time, or 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 you are familiar with the SQL language: the NEXT command acts as a combination of the SQL commands SELECT and FETCH, and allows you to use the structure of the data source to your advantage when retrieving data.)

Syntax **NEXT Command**

The syntax of the NEXT command is

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

where:

FOR

Is a prefix that is used with *int* or ALL to specify how many data source records are to be retrieved. If FOR is not specified, NEXT works like FOR 1 and the next record is retrieved. If the FOR phrase is used, the INTO phrase must also be used.

int

Is an integer constant or variable that specifies the number of data source records that are retrieved from the data source. Retrieval starts at the current position in the data source.

ALL

Specifies that starting at the current data source position, all data source segments referred to in the field list are examined.

path

Identifies the path to be read from the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical: simply specify the segment once. (For paths with multiple segments, if you wish to make the code clearer to readers, you can also specify segments between the anchor and target.)

To specify a segment, provide the name of the segment or of a field within the segment.

INTO

Is used with a stack name to specify the name of the stack into which the data source records are copied.

stack

Is the name of the stack that the data source values are placed into. Only one stack can be specified.

row

Is a subscript that specifies in which row of the stack the data source values are placed. If no subscript is provided, the data is placed in the stack starting with the first row.

where_expression1, where_expression2

Is any valid NEXT WHERE expression. You can use any valid relational expression, described in *Relational Expressions* in Chapter 8, *Expressions Reference*. NEXT can also use some enhanced screening conditions not available in other situations. For more information, see *Using Selection Logic to Retrieve Rows* on page 7-72.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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.

Subscripted Variables in WHERE Expressions

If you wish to use a subscripted variable (for example, `CustomerStack().CustID`) in the WHERE expression of a NEXT command, it is usually necessary to:

1. Assign the subscripted variable's value to a simple (that is, unsubscripted) variable immediately preceding the NEXT command.
2. Use the simple variable in the WHERE expression.

For example:

```
COMPUTE CompareCustID/A4 = CustomerStack().CustID;  
FOR ALL NEXT CustID TransCode INTO TransStack  
WHERE CustID EQ CompareCustID;
```

This restriction applies to relational data sources only; however, we recommend that you observe this restriction for all data sources in order to maximize the portability of your source code.

The WHERE expression in a NEXT command compares a data source value to a non-data source value. The non-data source value cannot be a subscripted variable if the NEXT command may be executed multiple times during the application. This means that you cannot use a subscripted variable for the non-data source value if the NEXT command is in a:

- REPEAT loop.
- Function that may be called multiple times.
- Procedure that may be called multiple times.

Example When Subscripted Variables Cannot be Used in WHERE Expressions

In the following source code, the NEXT command's WHERE expression is not valid because CustomerStack(Row).CustID is a subscripted variable that is executed in each iteration of the REPEAT loop:

```
COMPUTE Row/I4 = 1;
STACK CLEAR OrderStack;
REPEAT CustomerStack.FocCount;
    REPOSITION CustID;
    FOR ALL NEXT CustID OrderNum INTO OrderStack(OrderStack.FocCount+1)
        WHERE CustID EQ CustomerStack(Row).CustID;
ENDREPEAT Row = Row + 1;
```

The next example is also invalid, because CustomerStack(Row).CustID is a subscripted variable that is executed each time that the GetCustomer function is called:

```
MAINTAIN FILE CustData;
.
.
.
GetCustomer(Row);
.
.
.
CASE GetCustomer TAKES Row/I4;
    REPOSITION CustID;
    STACK CLEAR OrderStack;
    FOR ALL NEXT CustID OrderNum INTO OrderStack
        WHERE CustID EQ CustomerStack(Row).CustID;
ENDCASE
.
.
.
END
```

The simple way around this restriction is to assign a subscripted variable's value to a simple (that is, unsubscripted) variable immediately preceding the NEXT command, and then use the simple variable in the WHERE expression. In the following example, the WHERE expression is valid because the non-data source value is supplied by a simple variable, not by a subscripted variable:

```
COMPUTE Row/I4 = 1;
STACK CLEAR OrderStack;
REPEAT CustomerStack.FocCount;
    REPOSITION CustID;
    COMPUTE ThisCustomer/A4 = CustomerStack(Row).CustID;
    FOR ALL NEXT CustID OrderNum INTO OrderStack(OrderStack.FocCount+1)
        WHERE CustID EQ ThisCustomer;
ENDREPEAT Row = Row + 1;
```

Copying Data Between Data Sources

You can use the NEXT command to copy data between data sources. It is helpful to copy data between data sources when transaction data is gathered by one application and must be stored for use by another application. It is also helpful when the transaction data is to be applied to the data source at a later time or in a batch environment.

Example Copying Data to the Movies Data Source

For example, assume that you want to copy data from a fixed-format data source named FilmData into a FOCUS data source named Movies. You describe FilmData using the following Master File:

```
FILENAME=FILMDATA,   SUFFIX=FIX
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=RELEDATE,  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

When you are retrieving rows using the NEXT command, you have the option to restrict the rows you retrieve using the WHERE clause. The syntax for this option is

```
WHERE operand1 comparison_op1 operand2
[AND operand3 comparison_op1 operand4 ...]
operand1, operand2, operand3, operand4, ...
```

Are operands. In each NEXT WHERE expression, one operand must be a data source field, and one must be a valid Maintain expression that does not refer to a data source field.

For more information about Maintain expressions, see Chapter 8, *Expressions Reference*.

```
comparison_op1, comparison_op2, ...
```

Can be any of the comparison operators listed in *Logical Operators* in Chapter 8, *Expressions Reference* or any of the comparison operators listed in the following table. Some comparison operators may be listed in both places; this means that they can be used in a WHERE clause in an enhanced way.

Operator	Description	Example
IS, EQ, NE, IS_NOT	<p>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.</p> <p>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.</p>	WHERE ZipCode IS '112\$\$'

Operator	Description	Example
CONTAINS, OMITS	Select data source values that contain or omit a character string stored in a variable.	<pre> COMPUTE name/A4 = 'BANK'; FOR ALL NEXT bank_code bank_name into stackname WHERE bank_name CONTAINS name; returns all data where the word BANK is part of the bank name. COMPUTE name/A4 = 'BANK'; FOR ALL NEXT bank_code bank_name into stackname WHERE bank_name OMITS name; returns all data where the bank name does not include the word BANK. </pre>
EXCEEDS	Selects data source values that are greater than a numeric value.	
IN (<i>list</i>), NOT_IN (<i>list</i>)	Select data source values that are in or not in a list. IN and NOT_IN can be used with all data types.	<pre> FOR ALL NEXT emp_id bank_name INTO stackname WHERE bank_name NOT_IN ('ASSOCIATED BANK', CITIBANK) returns all data where the bank name is not in the list. </pre>

Operator	Description	Example
EQ_MASK, NE_MASK	<p>Select data source values that match or do not match a mask.</p> <p>Use the \$ sign to replace each letter in the value. Masks can only be used with alphanumeric data. The masked value may be hard coded or a variable:</p>	<pre>COMPUTE code/A4='AAA\$'; FOR ALL NEXT bank_name INTO stackname WHERE bank_ncode EQ_Mask code; returns all data where the bank code starts with AAA and has any character at the end. COMPUTE code/A4='AAA\$'; FOR ALL NEXT bank_name INTO Stackname WHERE bank_code NE_Mask code; returns all data where the bank code doesn't match the mask.</pre>

The following example retrieves every instance of the EmplInfo 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

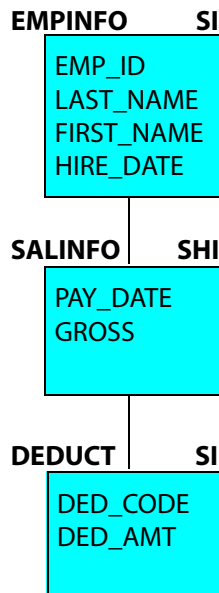
NEXT can also be used in conjunction with the MATCH command. This example issues a MATCH for employee ID. If there is not a match, a message is displayed. If there is a match, all the instances of the child segment for that employee are retrieved and placed in the stack Stackemp. The NEXT command can be coded as part of an ON MATCH condition, but it is not required, as the NEXT will only retrieve data based on the current position of higher level segments.

```
MATCH Emp_ID
ON NOMATCH BEGIN
    TYPE "The employee ID is not in the data source.";
    GOTO Getmore;
ENDBEGIN
FOR ALL NEXT Dat_Inc INTO Stackemp;
```

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 were not as many rows retrieved as you specified in the FOR prefix, you can determine how many rows were actually retrieved by checking the 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;
```

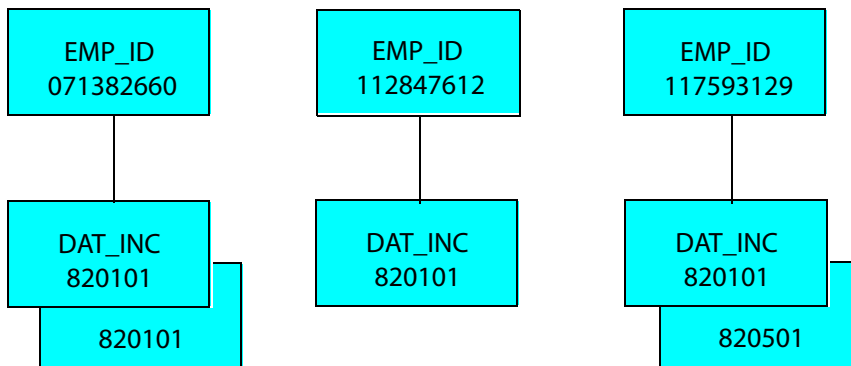
NEXT

When there is a possibility of short paths, and the intention is to retrieve the data from the parent even if there is no data for the child, NEXT should be used on one segment at a time, as described in *Data Source Navigation: NEXT Following NEXT or MATCH* on page 7-79. 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;
```

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 use NEXT through several segments, specify all the segments in one NEXT command or use several NEXT commands. If all of the segments are placed into one NEXT command, there is no way to know when data is retrieved from a parent segment and when it is retrieved from a child. To have control over when each segment is retrieved, each segment should have a NEXT command of its own. In this way, the first NEXT establishes the position for the second NEXT.

A NEXT command following a MATCH command works in a similar way: the first command (MATCH) establishes the data source position.

In the following example, the REPOSITION command places the position in the 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;
```

NEXT

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

Every NEXT command that uses a FOR prefix does so independent of any other NEXT command. If there are two NEXT commands, the first executes; when it is complete, the position is the last instance retrieved. The second NEXT command then executes and retrieves data from within the parent established by the first NEXT. In the following example, the first NEXT retrieves the first two instances from the 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 the stack.

```
STACK CLEAR Stemp;  
REPOSITION Emp_ID;  
FOR 2 NEXT Emp_ID INTO Stemp(1);  
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount);
```

The stack contains data from the following segments after the first NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2

The stack contains data from the following segments after the second NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2 and Pay_Date instance 1
3. Emp_ID instance 2 and Pay_Date instance 2
4. Emp_ID instance 2 and Pay_Date instance 3

The row in the INTO stack that the output is placed in is specified by supplying the row number after the stack name. When two NEXT commands are used in a row for the same stack, care must be taken to ensure that data is written to the appropriate row in the stack. If a stack row number is not specified for the second NEXT command, data is placed into the last row written to by the first NEXT, and existing data is overwritten. In order to place data in a different row, a row number or an expression to calculate the row number can be used. For example, the second NEXT command specifies the row after the last row by adding one to the variable FocCount:

```
FOR 2 NEXT Emp_ID INTO Stemp(1);
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount+1);
```

The stack now appears as follows. Notice that there is a new row 2.

1. Emp_ID instance 1
2. Emp_ID instance 2
3. Emp_ID instance 2 and Pay_Date instance 1
4. Emp_ID instance 2 and Pay_Date instance 2
5. Emp_ID instance 2 and Pay_Date instance 3

If a WHERE phrase is specified, the NEXT moves forward as many times as necessary to retrieve one record that satisfies the selection criteria specified in the WHERE phrase. For instance, the following retrieves the next record where the 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 syntax retrieves the next 3 records where the child segment's Gross field is greater than 1,000. As 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

Maintain allows separate segments to be joined in a one to one relation (among other ways). Unique segments are indicated by specifying a SEGTYPE of U, KU or DKU in the Master File, or by issuing a JOIN command. In a NEXT command, you retrieve a unique segment by specifying a field from the segment in the 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 the segment does not exist, the fields are treated as if they are fields in the parent segment. This means that the returned data is spaces, zeroes, and/or nulls (missing values), depending on how the segment is defined. In addition, the answer set contains as many rows as the unique segment's parent. 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.

You should query the FocFetch system variable in place of issuing the ON MATCH command; FocFetch accomplishes the same thing more efficiently. For more information, see *FocFetch* on page 7-43.

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 7-43.

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 7-43.

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 7-43.

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 *Terminating a Command's Syntax* in Chapter 6, *Language Rules Reference*.

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

```
PERFORM NextSet;
```

Reference Commands Related to PERFORM

- **CASE/ENDCASE** defines a Maintain function.
- **GOTO** transfers control to another function or to the end of the current function.

Using PERFORM to Call Maintain Functions

When you call a function as a separate statement (that is, outside of a larger expression), if the preceding command can take an optional semicolon terminator but was coded without one, you must call the function in a COMPUTE or PERFORM command. (You can use PERFORM for Maintain functions only, though not for Maintain functions that return a value.)

For example, in the following source code, the NEXT command is not terminated with a semicolon, so the function that follows it must be called in a PERFORM command:

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

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

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

For more information about terminating commands with semicolons, see *Terminating a Command's Syntax* in Chapter 6, *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 NotHere is performed after a NOMATCH condition occurs:

```
ON NOMATCH PERFORM NotHere;
```

Nesting PERFORM Commands

PERFORM commands can branch to functions containing other PERFORM commands. As each ENDCASE command is encountered, control returns to the command after the most recently executed PERFORM command. In this manner, control eventually returns to the original PERFORM.

Avoiding GOTO With PERFORM

It is recommended that you do not include a GOTO command within the scope of a PERFORM command. See *GOTO and PERFORM* on page 7-46 for information on the incompatibility of the PERFORM and GOTO commands.

RECOMPILE

If you compile a procedure under one release of FOCUS, and then run that procedure under a later release of FOCUS, the start up time of the first screen may be slow. In this case, Maintain displays a message informing you of the situation. To increase the start up speed of your procedure, recompile under the later release using the RECOMPILE command.

This command is outside the Maintain language, but is described here for your convenience. You cannot issue this command from within a Maintain procedure.

Syntax RECOMPILE Command

The syntax of the RECOMPILE command is

```
RECOMPILE procedure_name [AS newname]
```

where:

procedure_name

Is the name of the originally compiled procedure file.

newname

Is the name given to the recompiled procedure file. If you do not supply a name, the name of the newly-compiled procedure defaults to the name of the originally-compiled procedure.

Reference Commands Related to RECOMPILE

- **COMPILE** compiles a procedure, reducing the time needed to display its first Winform.
- **RUN** executes compiled procedures.

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 run. The value of *int* can be an integer constant, an integer field, or a more complex expression that resolves to an integer value. If you use an expression, the expression should resolve to an integer, although other types of expressions are possible. If the expression resolves to a floating-point or packed-decimal value, the decimal portion of the value will be truncated. If it resolves to an alphanumeric representation of a numeric value, it will be converted to an integer value.

Expressions are described in Chapter 8, *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 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 *Terminating a Command's Syntax* in Chapter 6, *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 run ten times, barring any conditions or commands to exit the loop. The ON NONEXT GOTO EXITREPEAT command causes the loop to be exited when there are no more instances of Sales in the data source. The COMPUTE command calculates TotSales within an execution of the loop. The ENDREPEAT command is the boundary for the loop. Commands after ENDREPEAT are not part of the loop. Because there is no loop counter, there is no way to know which repetition of the loop is currently executing:

```
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 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 run as long as the value of `A` is less than 3. It also initializes the counter `A` to 1. The second `REPEAT` command indicates that the nested loop will run until the value of `B` is greater than 4. It initializes the counter `B` to 1. Two `ENDREPEAT` commands are needed, one for each `REPEAT` command. Each `ENDREPEAT` increments its respective counters.

```
REPEAT WHILE A LT 3; A = 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.
- **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, starting at the beginning of the data source or path by first issuing the REPOSITION command is recommended.

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, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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.
- **MATCH** searches entire segments for a matching field value and can retrieve an exact match in the data source.

REVISE

The REVISE command reads a stack of transaction data and writes it to a data source, inserting new segment instances and updating existing instances.

REVISE combines the functionality of the INCLUDE and UPDATE commands. It reads each stack row and processes each segment in the specified path using the following logic:

```
MATCH key
ON MATCH UPDATE fields
ON NOMATCH INCLUDE segment
```

You specify a path running from an anchor segment to a target segment. For each segment in the path, REVISE matches the 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 using update fields, specify it using its segment name.

FROM

Indicates that the transaction data will be supplied by a stack. If this is omitted, the transaction data is supplied by the Current Area.

stack

Is the name of the stack whose data is being written to the data source.

row

Is a subscript that specifies the first stack row to be written to the data source. If omitted, it defaults to 1.

;

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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. Existing employee records are displayed in a grid. All of the information is stored in a stack named EmpStk.

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

CASE Alter_Data
FOR ALL REVISE Last_Name FROM EmpStk;
ENDCASE

END
```

When function Alter_Data is called from a form's trigger, 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.
- **UPDATE** updates data source fields.
- **COMMIT** makes all data source changes since the last COMMIT permanent.
- **ROLLBACK** cancels all data source changes made since the last COMMIT.

ROLLBACK

The ROLLBACK command processes a logical transaction. A logical transaction is a group of data source changes that are treated as one. The ROLLBACK command cancels prior UPDATE, INCLUDE, and DELETE operations that have not yet been committed to the data source using the COMMIT command.

Syntax **ROLLBACK Command**

The syntax of the ROLLBACK command is

```
ROLLBACK [;]
```

where:

```
;
```

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 run a compiled procedure by using the RUN command.

This command is outside the Maintain language, but is described here for your convenience. You cannot issue this command 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
```

Reference **Commands Related to Run**

- **MNTCON RUN** also executes compiled procedures.
- **COMPILE** and **MNTCON COMPILE** compile Maintain procedures, but we recommend that you use MNTCON COMPILE.

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, you should 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 an alphanumeric 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 on CMS, 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. For applications run from a terminal in an S/390 environment, the default device is the terminal.

Reference **Commands Related to SAY**

TYPE writes messages to a file or to a form.

Writing Segment and Stack Values

You can use the SEG and STACK prefixes to write the values of all 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.

SET

This command is outside the Maintain language, but is described here for your convenience, since many of these settings affect how Maintain behaves.

However, you can change a *limited* number of SET parameters from *within* a Maintain procedure using SYSMGR.FOCSET. For more information, see *SYS_MGR.FOCSET* on page 7-110.

For a list of SET parameters, see *Developing Applications*.

Syntax

How to Set Parameters

The syntax is

```
SET parameter = option[, parameter = option,...]
```

where:

parameter

Is the setting you wish to change.

option

Is one of a number of options available for each parameter.

You can set several parameters in one command by separating each with a comma.

You may include as many parameters as you can fit on one line. Repeat the SET keyword for each new line.

Note: This syntax is valid *only* in a WebFOCUS procedure!

Syntax **How to Set Parameters in a Request**

Many SET commands that change system defaults can be issued from within FOCUS TABLE requests. SET used in this manner is temporary, affecting only the current request. The syntax is

```
ON TABLE SET parameter value [AND parameter value ...]
```

where:

parameter

Is the system default you wish to change.

value

Is an acceptable value that will replace the default value.

Note: This syntax is valid *only* in a WebFOCUS report procedure!

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.

i

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 range in order to sort only part of the stack.

HIGHEST

If specified, sorts the stack in descending order. If not specified, the stack is sorted in ascending order.

column

Is a stack column name. At least one column name must be specified. The column must exist in the specified stack.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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 Maintain application. It can help developers ensure the most efficient interaction between Maintain and DBMS servers.

For use with Relational data sources, SYS_MGR can be used from within Maintain procedures to:

- Deactivate preliminary database operation checking by Maintain before an update and rely on the DBMS to perform its own internal checking, thus reducing processing time and resources (SYS_MGR.PRE_MATCH).
- Retrieve DBMS error codes, allowing developers to code applications to efficiently recover from error conditions (SYS_MGR.DBMS_ERRORCODE).
- Issue native SQL commands directly from a Maintain procedure (SYS_MGR.ENGINE).

The SYS_MGR syntax can be used from within a Maintain procedure to:

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

SYS_MGR.DBMS_ERRORCODE

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;
```

SYS_MGR.ENGINE

You can issue DBMS commands directly (SQL Passthru) from a Maintain procedure using the SYS_MGR.ENGINE command.

Note: Problems with direct commands are not reported in FOCERROR. You will need to use DBMS_ERRORCODE to determine the success or failure of these commands.

Syntax 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 Inserting a Row Into a Table (MS-SQL)

```
Compute rc/i8;  
Type "Inserting row into table MNTTAB2 "  
rc=sys_mgr.engine("SQLMSS","insert into mntbtb2  
values('X2','XDAT2222');");  
Type"ReturnCode=<<rc DBMS Err=<<SYS_MGR.DBMS_ERRORCODE";
```

You will need to test the return code to determine whether the record was inserted successfully (RC = 0).

If you are using MS-SQL, and the value you wanted to insert was a duplicate record, you would expect to see the following return codes:

```
Return Code= -1 DBMS Err= 2627
```

SYS_MGR.FOCSET

Using SYS_MGR.FOCSET, you can set certain environment settings. See *Developing Applications* for a complete description of the environment settings.

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 DEFCENT From a Maintain Procedure**

The following code

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

produces the following output:

```
After setting DEFCENT=21, MYDATE=2190/01/01
```


Example Setting PASS From a Maintain Procedure

The following code will set the password to DBAUSER2:

```
SYS_MGR.FOCSET( 'PASS', 'DBAUSER2.' );
```

Note: When setting a password for DBA access, keep in mind that the last value set from within the application will be in effect for all transactions for that end user's session.

SYS_MGR.PRE_MATCH

By default, Maintain first ensures a database row exists before it updates or deletes it and ensures a database row does NOT exist before including a new row. For example, when Maintain processes an INCLUDE it first issues:

```
SQL SELECT keyfld FROM tablename WHERE keyfld = keyvalue;
```

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.PRE_MATCH. For high volume transactions this can positively affect performance.

Note:

- Since you are not checking to see if the row exists or not, it is important to write code that catches errors by inspecting FOCERROR.
- The PRE_MATCH setting is local to the current MAINTAIN procedure. Changing it does not change the value in the parent procedure or in subsequently called procedures.

Syntax **Setting PRE_MATCH**

The syntax is

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

or

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

where:

0

Disables prematching.

1

Turns on prematching.

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

```
SYS_MGR.GET_PRE_MATCH();
```

or

```
SYS_MGR.PRE_MATCH;
```

Example **Setting PRE_MATCH Off**

Suppose you have a Maintain procedure with the following code:

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

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

TYPE

The TYPE command displays messages in a TYPE message box (if the procedure has Winforms), writes them to the screen, or to a sequential file (if the command names a file). You can use TYPE to trace application flow-of-control and record an accounting trail.

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 DYNAM or FILEDEF command) prior to the first usage. The message string can be up to 256 characters in length. The output starts in column 1. In order to append to an existing file or to write to a file from more than one procedure, append to the file by specifying the appropriate option in the DYNAM or FILEDEF command.

If ON *ddname* is not provided and the procedure includes Winforms, the message string is displayed in a TYPE Winform.

message

Is the information to be displayed or written. The message must be enclosed in double quotation marks ("). The message can contain:

- Any literal text.
- Variables.
- Horizontal spacing information.
- Vertical spacing information.

The layout of the message is exactly what is specified.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *Language Rules Reference*.

Reference **Commands Related to TYPE**

SAY writes messages to a file or to the default output device; messages can include multiple expressions of all types.

Including Variables in a Message

You can embed variables in a message by prefixing the variable with a left caret (<). Unless the field name is the last item in the string, it must be followed by a space. Maintain does not include the caret and space in the display. For example:

```
TYPE "Accepted: <Indata(Cnt).Fullname";
```

Embedding Horizontal Spacing Information

TYPE information can be placed in a specific column or can be moved a number of columns away from the current position. The following example

```
TYPE "<20 This starts 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 alphanumeric 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.

fields

Is used to specify which data source fields to update. You must specify every field that you wish to update. You cannot update key fields. All fields must be in the same path.

FROM

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

stack

Is the name of the stack whose data is used to update the data source. Only one stack can be specified.

row

Is a subscript that specifies the first stack row to use to update the data source.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *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.
- When an UPDATE command is complete, the variable FocError is set. If the UPDATE is successful, FocError is set to zero. If the records do not exist, and are therefore unchanged, FocError is set to a non-zero value and—if the UPDATE is set-based—FocErrorRow is set to the number of the row that failed.
- Maintain requires that data sources to which it writes have unique keys.

Reference Commands Related to UPDATE

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

Update and Transaction Variables

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

Example Using Stacks

In the following example, the user enters many employee IDs and new names at one time. Rather than performing a MATCH on each row in the stack, this function checks FocError after the UPDATE command. If FocError is zero, a COMMIT is issued and the function is exited. If FocError is not zero, another function, which tries to clean up the data, is performed. The IF command, which starts at the beginning of the function checks to see whether there are any rows in the stack. If the stack does contain have any rows, a form is displayed allowing the user to enter new data. If the stack contains rows, the user has made a mistake, so a different form is displayed allowing the user to edit the entered data.

The Maintain procedure contains:

```

STACK CLEAR Namechg;
PERFORM Chngname;
CASE Chngname
IF Namechg.FocCount LE 0
    THEN WINFORM SHOW Myform1;
    ELSE WINFORM SHOW Myform2;
FOR ALL UPDATE Last_Name FROM Namechg;
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. Winforms 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 (controls are sometimes referred to as Winform objects).
- 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. 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

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.

HIDE

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.

UNHIDE

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 end user can unhide a partially covered form by clicking it with the ONTOP function key (PF24).

REFRESH

Repopulates the form's data values as if control had returned to the form from a trigger, but without making the form active.

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 popup forms from the screen. Closed non-popup Winforms remain on the screen until hidden by a WINFORM HIDE command or the end of the current Maintain procedure.

STOP

Closes all Winforms. Terminates the Winform environment. Directs TYPE messages to the screen.

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.

Except where noted otherwise in *Dynamically Changing Winform Control Properties* on page 7-125, properties can be set for all types of controls.

property

Is any valid property. Properties are described in *Dynamically Changing Winform Control Properties* on page 7-125.

value

Is a value that is valid for the specified property. Properties and their values are described in *Dynamically Changing Winform Control Properties* on page 7-125.

variable

Is any scalar variable—a user-defined field or a stack cell—to which you will assign the value of the specified property of the specified form or control.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating a Command's Syntax* in Chapter 6, *Language Rules Reference*.

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.
- **TYPE** displays messages on the screen or writes them to a file.

Managing the Flow of Control in a Winform

When a WINFORM command is encountered that contains the SHOW option, control is passed to the named Winform. Control does not return to the procedure until the user exits the Winform. While the Winform is active, controls within the Winform can call Maintain functions (controls are also known as Winform objects, and functions are also known as cases). This is accomplished by the use of triggers that specify what happens when the associated event occurs.

The following is an example of a triggered Maintain function. Two Winforms are hidden, a background Winform is displayed, and the main Winform is made active. The background Winform is just a background graphic, so there is no need to activate it:

```
WINFORM HIDE Addr;
WINFORM HIDE Info;
WINFORM SHOW_INACTIVE Back;
WINFORM SHOW_ACTIVE Main;
```

The next example uses the WINFORM command after a MATCH in a triggered function. If there is a match on Emp_ID, two WINFORM commands are performed. Empinput, the Winform used to enter the employee's ID number, is hidden and EmpInfo, the Winform that displays information for the employee whose ID was found, is activated. If the MATCH is unsuccessful, the procedure displays the NotFound Winform which informs the application user that the employee ID was not found. It does not hide the Empinput Winform because the application's user should be able to see the unsuccessful employee ID in case the error was caused by a typing mistake:

```
CASE Findemp
MATCH Emp_ID;
ON MATCH BEGIN
    WINFORM HIDE Empinput;
    WINFORM SHOW EmpInfo;
ENDBEGIN
ON NOMATCH WINFORM SHOW Notfound;
ENDCASE
```

A trigger action can be a case (also known as a function) or a system action (such as Exit).

When a trigger that specifies a function is called, the function is performed. The Winform remains visible to the end user but does not accept keyboard activity. When the function or functions finish processing, the Winform again becomes available to end user activity.

A trigger can display another Winform. This is done by specifying a WINFORM command within the triggered Maintain function. In this situation, if the Winform in the function is made active by the SHOW or SHOW_ACTIVE option, the Winform that triggers the function continues to be displayed but is no longer active. When the second Winform is exited, control returns to the function that called it. When that function is exited, control returns to the Winform from which the trigger was called, and at that time the calling Winform becomes active again. The following are the steps in a generic example:

1. Winform A is active, and the user invokes a trigger.
2. The trigger calls a function.
3. The function displays Winform B and makes B active. Winform A is no longer active.
4. Once Winform B is exited, the rest of the function is performed.
5. When the function is exited, Winform A becomes active again.

In this example, if Winform B has a trigger that issues a WINFORM SHOW A command, a warning message is displayed because SHOW cannot be specified for a Winform that is already displayed to the user.

Displaying Default Values in a Winform

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
	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 Winform 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. (Controls are also known as Winform objects.) You can use these commands with all Winform controls. (For some properties, such as a grid's current column, you call a function instead of using WINFORM SET and GET.)

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

You can set and query the following properties:

- **Color: controls in general.** You can change the foreground and background colors of a Winform control, or of the entire Winform, by setting the FOREGROUND_COLOR and BACKGROUND_COLOR properties respectively. You can set these properties to a color's name or to its corresponding numeric value; the WINFORM GET command retrieves the numeric value. The following colors are available:

Colors	Numeric Values
BLACK	1
BLUE	2
GREEN	3
TURQ	4
RED	5
PINK	6
YELLOW	7
WHITE	8

For each control, you can change the foreground color or the background color, but not both at the same time: when you set one, the other is automatically set to black.

For example, the following command changes the Sales entry field in the SalesSummary Winform to red:

```
WINFORM SET SalesSummary.Sales.BACKGROUND_COLOR TO RED;
```

- **Color: grid columns.** You can change the background and foreground colors of a grid column using the ChangeColBcolor and ChangeColFcolor functions respectively. The syntax for calling these functions is

```
[COMPUTE] formname.gridname.ChangeColBcolor(ColumnNumber, ColorNumber);  
[COMPUTE] formname.gridname.ChangeColFcolor(ColumnNumber, ColorNumber);
```

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 the COMPUTE keyword is unnecessary.

formname

Is the name of the Winform that contains the grid.

gridname

Is the name of the grid.

ColumnNumber

Is the column number in the grid.

ColorNumber

Is the number associated with the color you want in the grid column. Colors and numbers are listed in the previous table.

- **Visibility: general.** You can display and hide a control by setting its `VISIBLE` property to `YES` or `NO` respectively. The corresponding values retrieved by `WINFORM GET` are 1 and 0.

For example, the following command hides the Salary entry field in the EmployeeReview Winform:

```
WINFORM SET EmployeeReview.Salary.VISIBLE TO NO;
```

If you wish to dynamically hide an entry field's prompt, you must issue a separate `WINFORM SET` command that includes the `STATIC` keyword. The syntax is

```
WINFORM SET formname.fieldnamesSTATIC.VISIBLE TO NO;
```

For example, the following command hides the Salary entry field's prompt in the EmployeeReview Winform:

```
WINFORM SET EmployeeReview.SalarySTATIC.VISIBLE TO NO;
```

- **User control: general.** You can protect and unprotect a control from being used by setting `PROTECTED` to `YES` and `NO` respectively. The corresponding values retrieved by `WINFORM GET` are 1 and 0.

When a control is protected, an application user cannot tab to it or use it. You cannot unprotect control types that, by definition, cannot be tabbed to or manipulated, such as text.

For example, the following command protects the CustomerNameList list box in the CustomerForm Winform:

```
WINFORM SET CustomerForm.CustomerNameList.PROTECTED TO YES;
```

When a control is protected using the `Protected` property, it cannot be made available for user control by setting `PROTECTED` to `NO`. If you protect a control by selecting its `Protected` check box, the `WINFORM SET PROTECTED TO NO` command cannot affect or eliminate the control's protection.

- **User control: adding grid columns.** You can prevent a user from adding rows to a grid using the `SetStackMode` function. The user is still able to scroll through the stack and update and delete data in the stack's existing rows. (If you also wish to prevent users from changing existing data, you can protect the stack in the Winform Painter.)

The syntax for calling this function is

```
[COMPUTE] formname.gridname.SetStackMode(ADDOFF);
```

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 the COMPUTE keyword is unnecessary.

formname

Is the name of the Winform that contains the grid.

gridname

Is the name of the grid.

ADDOFF

Prevents the user from placing the cursor below the last line of the grid that contains data.

- **Input focus.** You can specify which control is active—that is, which control is the focus of user activity such as pressing a function key—by setting its FOCUS property. The control to receive focus is always set to the value HERE. The WINFORM GET command retrieves the name of the control that currently has focus.

For grids, the row indicated by the current value of `FocIndex` receives focus. You cannot give focus to control types that you cannot tab to, such as text.

For example, the following command makes the `NextButton` push button in the `CustomerForm` Winform the active control:

```
WINFORM SET CustomerForm.NextButton.FOCUS TO HERE;
```

The following command assigns the name of the control that is currently active (that is, that currently has focus) to the variable `ControlHasFocus`:

```
WINFORM GET CustomerForm.FOCUS INTO ControlHasFocus;
```

- **Bold font for a control's text.** You can make a control's text bold, or not bold, by setting its `BOLD_FONT` property to YES or NO respectively. The corresponding values retrieved by WINFORM GET are 1 and 0.

For example, the following command bolds the text in the JobTitleList list box in the EmployeeReview Winform:

```
WINFORM SET EmployeeReview.JobTitleList.BOLD_FONT TO YES;
```

- **Underline font for a control's text.** You can make a control's text underlined, or not underlined, by setting its `UNDERLINE_FONT` property to YES or NO respectively. The corresponding values retrieved by WINFORM GET are 1 and 0.

For example, the following command underlines the text in the SubmitButton button in the EmployeeReview Winform:

```
WINFORM SET EmployeeReview.SubmitButton.UNDERLINE_FONT TO YES;
```

- **Blinking font for a control's text.** You can make a control's text blink, or not blink, by setting its `BLINK_FONT` property to YES or NO respectively. The corresponding values retrieved by WINFORM GET are 1 and 0.

For example, the following command makes the text in the HeadingText banner blink at the top of the EmployeeReview Winform:

```
WINFORM SET EmployeeReview.HeadingText.BLINK_FONT TO YES;
```

- **Current grid column and row.** You can determine and set which grid column and row are current (that is, active) using several variables defined for grids. These enable you to refer to the current grid column and row either by counting from the first column or row in the grid (that is, an absolute reference), or by counting from the first column or row currently visible in a grid, which enables you to take into account how the grid has been scrolled (that is, a relative reference—relative to the first currently visible column).

- **CurStkRowNum.** The CurStkRowNum variable always contains the value of the current row number in the grid, and is set automatically by Maintain. This system variable is helpful for determining the current row number. For example, the following command assigns the current row number of Grid1 to the variable RowNumber:

```
RowNumber = ContactListForm.Grid1.CurStkRowNum;
```

Maintain automatically sets the FocIndex system variable to the value of the CurStkRowNum variable when the user leaves the grid and clicks on another control on the Winform. The syntax for this variable is

```
formname.gridname.CurStkRowNum
```

where:

formname

Is the name of the Winform.

gridname

Is the name of the grid.

- **CurStkColNum.** The CurStkColNum variable always contains the value of the current column number in the grid, and is set automatically by Maintain. This variable is helpful for determining the current column number. For example, the following command assigns Grid1's current column number to the variable ColNumber:

```
ColNumber = ContactListForm.Grid1.CurStkColNum;
```

The syntax for this variable is

```
formname.gridname.CurStkColNum
```

where:

formname

Is the name of the Winform.

gridname

Is the name of the grid.

- **CurGrdRowNum.** The CurGrdRowNum variable contains the number of the current grid row, relative to the first row that is visible in the grid, and is set automatically by Maintain. This variable is different from CurStkRowNum if you scroll the grid. For example, if you scroll down a grid, your current row number may be 12, but the relative row number may be 3.

The syntax for this variable is

formname.gridname.CurGrdRowNum

where:

formname

Is the name of the Winform.

gridname

Is the name of the grid.

- **CurGrdColNum.** The CurGrdColNum variable contains the number of the current grid column, relative the first column that is visible in the grid, and is set automatically by Maintain. This variable is different from CurStkColNum if you scroll the grid left or right.

The syntax for this variable is

formname.gridname.CurGrdColNum

where:

formname

Is the name of the Winform.

gridname

Is the name of the grid.

WINFORM

CHAPTER 8

Expressions Reference

Topics:

- Types of Expressions You Can Write
- Writing Numeric Expressions
- Writing Date Expressions
- Writing Date-Time Expressions
- Writing Alphanumeric 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.

Expressions in Maintain are similar to expressions in other FOCUS facilities, but some behavior and rules differ, and some functionality is enhanced.

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 8-3.

- **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 Expressions* on page 8-8.
- **Date-time expressions.** For details, see *Writing Date-Time Expressions* on page 8-14.
- **Alphanumeric.** Use an alphanumeric expression to manipulate alphanumeric 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$$$$$$$$') ;
```

An alphanumeric expression returns an alphanumeric value. For details see *Writing Alphanumeric Expressions* on page 8-19.

- **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 8-22.
- **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 8-25.

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 an alphanumeric expression to concatenate a first name and last name and assign it to the variable FullName, make sure you define the variable as alphanumeric.

Writing Numeric Expressions

A numeric expression performs a calculation that uses numeric constants, variables, operators, or functions to return a number. A numeric expression can consist of the following components, 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	<i>DIV</i>	Native, except converts packed decimal to double-precision floating point
Remainder Division	<i>MOD</i>	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	Alphanumeric

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

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

When a date is compared to a number, the number is converted to integer and the comparison is performed. However, if you are not using date format, but instead an alphanumeric, integer, or packed-decimal format with date-edit options (sometimes referred to in the Information Builders user community as “old dates”), the date is converted to date format prior to the comparison.

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 must 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 *Developing Applications*.

Writing Date Expressions

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

A date expression can consist of the following components, 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;
```

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 for all calculations and displays. 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, formatted as single components such as a day of the week or year (formats D, W, Y, and YY), cannot be meaningfully converted to a more complete date (such as a year with a month). Of course, you can use these date variables in same-type date expressions.

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

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

The following conversion hierarchy applies to date formats:

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 addition is performed:

- A date plus a number yields a date.
- A number plus a date yields a date.

It is up to the user to make sure the expression yields a meaningful result.

When subtraction is performed:

- A date cannot be subtracted from a number.
- A date minus a number results in a value with the same format as the date.
- When a date with format M or Q is subtracted from a higher type of date, the operand on the right is converted to the format of the operand on the left.
- When a two-component date is subtracted from a three-component date, or vice versa, the variable with the lesser format is promoted to the type of the variable with the higher format.
- When subtracting a Q format date from an M format date, or vice versa, the operand on the right is converted to the same format as the operand on the left, and the result is an integer.

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;
```

Writing Date-Time Expressions

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

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

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 the *Describing Data* manual. 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.

Example Using a Date-Time Value in a COMPUTE Command

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

Manipulating Date-Time Values Directly

The only direct operations that can be performed on date-time variables and constants are comparison using a logical expression and simple assignment of the form $A = B$. All other operations are accomplished through a set of date-time subroutines. For more information see *Writing Alphanumeric Expressions* on page 8-19.

Comparing and Assigning Date-Time Values

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

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

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

Assignments are permitted between date-time formats of equal or different lengths. Assigning a 10-byte date-time value to an 8-byte date-time value truncates the microsecond portion (no rounding takes place). Assigning a short value to a long one sets the low-order three digits of the microseconds to zero.

Other operations, including arithmetic, concatenation, and the reporting operators EDIT and LIKE on date-time operands are not supported. Reporting prefix operators that work with alphanumeric fields are supported.

Example Testing for Missing Date-Time Values

Consider the DATETIM2 Master File:

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

Field DT1 supports missing values. Consider the following request:

```
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
HCNVRT	Converts date-time values to alphanumeric format for use with operators such as EDIT, CONTAINS, and LIKE.
HINPUT	Converts an alphanumeric string to a date-time value.
HADD	Increments date-time values by a specified number of units.
HDIFF	Returns the number of units of a specific date-time component between two date-time values.
HNAME	Extracts specified components of a date-time value and converts them to alphanumeric format.
HPART	Extracts a component of a date-time value in numeric format.
HSEPTPT	Inserts the numeric value of a specified component in a date-time field.
HMIDNT	Changes the time portion of a date-time field to midnight.

Function Name	Description
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.

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

Reference Notes Regarding ISO Standard Date-Time Representations

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

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

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

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

Writing Alphanumeric Expressions

An alphanumeric expression manipulates alphanumeric constants, variables, concatenation operators, or functions to return an alphanumeric value.

An alphanumeric expression can consist of the following components, shown below in **bold**:

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

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

- An alphanumeric variable. For example:

```
COMPUTE AddressPartTwo = STATE ;
```

- A function returning an alphanumeric result. For example:

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

- Two or more alphanumeric 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 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 Alphanumeric Expressions

Any non-alphanumeric expression that is embedded in an alphanumeric expression is automatically converted to an alphanumeric 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 alphanumeric escape character. You can use it to:

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

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

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 Extracting Substrings and Using Strong and Weak Concatenation

The following example shows how to use the MASK 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:

```
COMPUTE FILE EMPLOYEE
FIRST_INIT/A1 = MASK (FIRST_NAME, '9$$$$$$$$');
NAME/A19 = LAST_NAME || (',' | FIRST_INIT | '.');
END
```

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 MASK 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, strong concatenation suppresses the trailing blanks.

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 alpha_operator alphanumeric_constant  
numeric_expression numeric_operator numeric_constant
```

where:

alpha_operator

Can be any of the following: EQ, NE, OMMITS, 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 8-25. You can also assign the result of a Boolean expression to a numeric or alphanumeric 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 an alphanumeric variable, it may have the values TRUE, FALSE, 1, or 0; TRUE and 1 are equivalent, as are FALSE and 0. A numeric variable may have the values 1 or 0.

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

```
IF NAME EQ 'JOHN DOE'
```

OR cannot be used between constants in a relational expression. For example, the following expression is not valid

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

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
Tests for and selects values that include a character string matching a test value	CONTAINS
Tests for and selects values that do not include a character string matching a test value	OMITS
Inequality	NOT
Compound expression	AND
Equality	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 8-23.

expression

Is a numeric, alphanumeric, 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
Alphanumeric	space	null

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

Null values affect the results of expressions that perform aggregating calculations such as averaging and summing. See the topics about null data and missing data in *Describing 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 a variable that was defined with the MISSING attribute in its Master File or, for user-defined variables, in its COMPUTE command.

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 whether a value is null is

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

Example Testing Null Values

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

```
IF Returns EQ MISSING THEN BEGIN
.
.
.
ENDBEGIN
```

CHAPTER 9

Modifying Data Sources With MODIFY

Topics:

- Introduction
- Examples of MODIFY Processing
- Additional MODIFY Facilities
- Describing Incoming Data
- Special Responses
- Entering Text Data Using TED
- Modifying Data: MATCH and NEXT
- Computations: COMPUTE and VALIDATE
- Messages: TYPE, LOG, and HELPMESSAGE
- Case Logic
- Multiple Record Processing
- Advanced Facilities
- MODIFY Syntax Summary

These topics describe how to maintain FOCUS-supported data sources using the FOCUS MODIFY facility. MODIFY requests can add, update, and delete data from FOCUS data sources, including HOLD files converted to FOCUS format (see the *Creating Reports* manual).

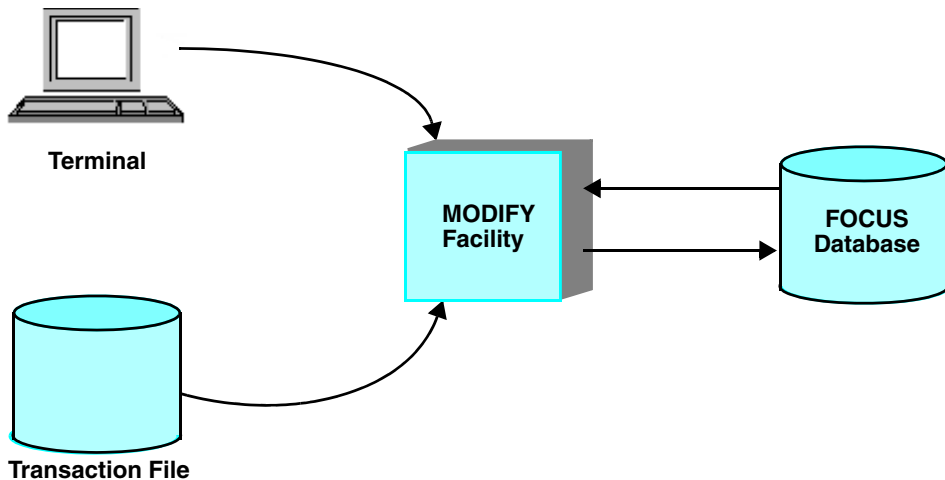
The MODIFY facility is also used to maintain data in relational structures, Adabas data sources, and VSAM data sources. See documentation for specific data adapters for details about using MODIFY in those environments.

Introduction

A MODIFY request processes a transaction in three steps:

1. It reads a transaction for incoming data values. Transactions can come from external data sources, may be supplied by the user in screens or in response to prompts, or can be included as part of the request itself.
2. It selects a segment instance for changing or deleting, or confirms that a segment instance does not exist yet in the data source.
3. It changes or deletes the segment instance it selected, or adds a new segment instance.

This is shown graphically in the following diagram:



The request first reads a transaction (that is, a related collection of incoming data values). *Describing Incoming Data* on page 9-19 describes the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements that describe transactions read by the request.

After it reads a transaction, the request selects a segment instance in the data source to modify. It does this in either of two ways:

- It searches the data source for segment instances containing the same values as the transaction. This is done with a MATCH statement.
- It selects the next segment instance after the current position. This is done with a NEXT statement.

The MATCH and NEXT statements are discussed in *Modifying Data: MATCH and NEXT* on page 9-58.

The request then either adds, updates, or deletes data source values using the incoming values, or it rejects the transaction.

Examples of MODIFY Processing

This section provides examples of MODIFY processing that add, update and delete data from a data source.

Adding Data to a Data Source

The following sample MODIFY request adds new employee data to the EMPLOYEE data source. When you run the request, it prompts you for an employee ID number, last name, and first name. After you enter these three values, the request adds the information to the data source and prompts you for three more values for the same fields. When you are finished entering data, end execution by entering the word END to any prompt.

The request is as follows:

```
1. MODIFY FILE EMPLOYEE
2. PROMPT EMP_ID LAST_NAME FIRST_NAME
3. MATCH EMP_ID
4.     ON MATCH REJECT
5.     ON NOMATCH INCLUDE
6. DATA
```

The parts of the request are as follows:

1. The MODIFY FILE EMPLOYEE statement indicates that the request modifies the EMPLOYEE data source.
2. The PROMPT statement indicates that the request will prompt you for the employee's ID (EMP_ID), last name, and first name on the terminal.
3. The MATCH EMP_ID statement searches the data source for the employee ID that you entered.
4. If the ID is already in the data source (that is, an ID in the data source matches the ID you entered), the MATCH statement rejects your transaction.
5. If the ID is not yet in the data source, the MATCH statement adds your transaction to the data source.
6. The DATA statement begins prompting for data.

Procedure Updating Data in a Data Source

MODIFY requests can update data in a data source, replacing data source values with transaction (incoming data) values. The following sample request updates employee department assignments and salaries. When you run the request, it reads the data from a separate data source called EMPDEPT. Each record in the data source consists of three fields:

- The EMP_ID field contains the employee ID number. It is the first nine characters on the record.
- The DEPARTMENT field contains the new department assignment, and is the next ten characters.
- The CURR_SAL field contains the new salary, and is the last eight characters.

This is the EMPDEPT data source:

```
* * * TOP OF FILE * * *  
071382660PRODUCTION27500.00  
112847612SALES      24800.75  
451123478MARKETING 26950.00  
* * * END OF FILE * * *
```

The request is as follows:

```
MODIFY FILE EMPLOYEE  
1. FIXFORM EMP_ID/9 DEPARTMENT/10 CURR_SAL/8  
  
2. MATCH EMP_ID  
2.     ON NOMATCH REJECT  
2.     ON MATCH UPDATE DEPARTMENT CURR_SAL  
3. DATA ON EMPDEPT  
4. END
```

The parts of the request are as follows:

1. The FIXFORM statement indicates that the transaction records are in fixed positions in the EMPDEPT data source and describes the positions of the fields in each record.
2. The MATCH EMP_ID statement searches the data source for the employee ID in each record. If the ID is not in the data source, the request rejects the record. If the ID is in the data source, the request replaces the DEPARTMENT and CURR_SAL values in the data source with the values on the record.
3. The DATA statement indicates that the data is contained in the data source EMPDEPT. EMPDEPT is the ddname to which the data file is allocated, and can be different from the system file name.
4. The END statement completes the request and initiates processing.

Procedure Deleting Data From a Data Source

This sample request deletes information on employees from the data source. When you run the request, it prompts you for an employee ID. When you enter the ID, it deletes all information relating to that employee from the data source.

```
MODIFY FILE EMPLOYEE
1. PROMPT EMP_ID
2. MATCH EMP_ID
   ON MATCH DELETE
   ON NOMATCH REJECT
3. DATA
```

The parts of the request are as follows:

1. The PROMPT statement indicates that the request will prompt you for the employee's ID.
2. The MATCH statement searches for the employee ID in the data source. If the ID is in the data source, the request deletes all information relating to the employee from the data source.
3. The DATA statement begins prompting for data.

The above examples show how to add, update, and delete data from a data source. Each request indicates the data source it is modifying, the method of reading data, the transaction values it searches for in the data source, and the actions it takes depending on whether the values are in the data source or not. If it is reading a transaction data source, the request must indicate the name of the data source.

Additional MODIFY Facilities

You can also instruct the request to perform other tasks:

- Test transaction values to determine whether they are acceptable. You do this using the VALIDATE statement, described in *Computations: COMPUTE and VALIDATE* on page 9-92.
- Perform calculations and store the results in either transaction or temporary fields. You do this using the COMPUTE statement, described in *Computations: COMPUTE and VALIDATE* on page 9-92.
- Display messages that contain values from transaction fields, temporary fields, or data source fields. You do this using the TYPE statement, discussed in *Messages: TYPE, LOG, and HELPMESSAGE* on page 9-117.
- Record transactions processed by the request using the TYPE and LOG statements described in *Messages: TYPE, LOG, and HELPMESSAGE* on page 9-117. These statements can sort accepted transactions from rejected transactions and can sort rejected transactions by reason for rejection.

You can design MODIFY requests using Case Logic, a method which divides requests into sections called “cases.” The request can branch to the beginning of a case during execution. Case Logic, discussed in *Case Logic* on page 9-132, makes it possible for requests to offer the terminal operator selections and to process transactions in different ways.

You can design MODIFY requests that process multiple segment instances at one time. Multiple Record Processing is described in *Multiple Record Processing* on page 9-158, including the modification of several segment instances on one FIDEL screen.

Reference Notes on Using JOIN Syntax With MODIFY

For software that supports the MODIFY facility, note the following:

- The JOIN command allows you to read (but not to modify) data in a second FOCUS data source using the MODIFY LOOKUP function. To modify multiple FOCUS data sources in one request, use the COMBINE command.
- The LOOKUP function in MODIFY requests cannot be used on a DEFINE-based JOIN; DEFINE is not evaluated during a MODIFY procedure.
- The MODIFY LOOKUP function cannot retrieve data in a cross-referenced segment using concatenated fields (a multi-field join).

FOCUS offers a variety of other advanced features that facilitate use of the MODIFY command in more complex applications. These features are listed below and described in *Advanced Facilities* on page 9-189:

- The COMBINE command for modifying multiple FOCUS data sources in one MODIFY request.
- The COMPILE command for translating MODIFY requests into compiled code ready for execution.
- The ACTIVATE and DEACTIVATE statements for activating and deactivating fields.
- The Checkpoint and Absolute File Integrity facilities and the COMMIT and ROLLBACK Subcommands for protecting FOCUS data sources from system failures.
- The ECHO facility for displaying the logical structure of MODIFY requests.
- Dialogue Manager system variables that record execution statistics every time a MODIFY request is run.
- FOCUS query commands that display statistical information on MODIFY request executions and FOCUS data sources.

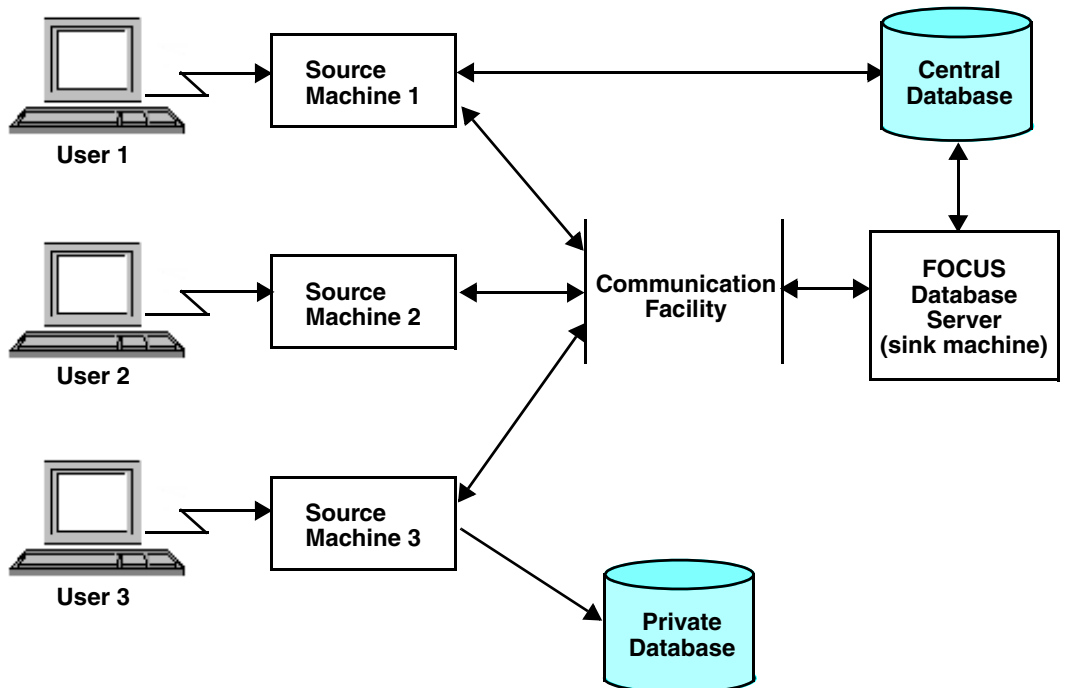
The rest of this introduction contains:

- The basic syntax of MODIFY requests.
- Instructions for executing MODIFY requests.
- A summary of facilities other than MODIFY that can be used to maintain FOCUS data sources.
- A short description of the parts of the EMPLOYEE data source most used in the examples.

Multiple User Access

Suppose you need to update a particular data source, but three other users have been assigned to work on the data source at the same time. How can you be sure that one user's changes will not override or overwrite another user's changes? MODIFY, used in conjunction with the Simultaneous Usage (SU) facility, ensures data integrity under those circumstances.

To enter SU mode, you initiate a background job process called a FOCUS Database Server. The user ids running FOCUS or Host Language Interface programs are called source machines. The users (using their source machines) send requests and transactions to the FOCUS Database Server, which processes the transactions and transmits the retrieved data or messages back to the source machine. The following diagram illustrates the process:



Under SU, when you run a MODIFY request

1. The request identifies the instance to be changed with MATCH or NEXT commands.
2. The source machine forwards the transaction values to the FOCUS Database Server, which uses the values to retrieve the correct instance.
3. The FOCUS Database Server retrieves the original data source instance, holds one copy, and sends another to the source (user id) that requested the data.

4. The source machine updates its copy of the instance with the new field values, or marks the copy for deletion and sends the updated copy back to the FOCUS Database Server. The FOCUS Database Server compares the copy of the instance that it saved with the instance stored in the data source to check whether the data source instance has since been updated by another user.

At this point, two courses of action are possible:

- If the copy and the current instance in the data source are the same, FOCUS changes the instance using the copy from the source machine.
- If the original and the current instance in the data source are different, SU signals a conflict and rejects the source machine copy.

Notice that a source machine may work on separate, locally controlled data sources.

Reference **SU Features**

With SU you can display a list of the active source machine userids and the fields of the FOCUS Database Server data sources from your source machine, and record all user actions in a sequential data source called HLIPRINT. The HLIPRINT data source records each user action, the data source on which the action took place, the segment read or modified by the action, and the user id that issued the action. It can also include the:

- Date and time of the action.
- CPU time it took to execute the action.
- Number of I/O operations required to execute the action.
- Name of the FOCUS stored procedure executing the action, and the name of the case executing the action (for MODIFY requests using Case Logic).

Another SU feature is the FOCURRENT variable that alerts users to transaction conflicts. When you submit a MODIFY transaction in SU, FOCUS stores a value in a variable called FOCURRENT to indicate what happened to the transaction. You can design your MODIFY requests to test FOCURRENT and take different actions, depending on whether the transaction was accepted or rejected. The following request tests the FOCURRENT variable:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL
CASE NEWSAL
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT CURR_SAL
    ON MATCH UPDATE CURR_SAL
    ON MATCH IF FOCURRENT EQ 0 GOTO TOP;
    ON MATCH TYPE
        "VALUE CHANGED. NEW VALUE <D.CURR_SAL>"
ENDCASE
DATA
```

The request prompts for an employee ID and then branches to the case NEWSAL. If the ID is in the data source, you are prompted for the current salary of the employee; the current salary is updated on the source machine copy. The transaction is submitted.

Next, the request tests the values of the variable FOCURRENT:

- If FOCURRENT is 0, the transaction is accepted and the request prompts you for the next EMP_ID.
- If FOCURRENT is not 0, the transaction is rejected. The request branches back to the top of the procedure. If the instance is found, FOCUS prompts for the current salary and resubmits the transaction. If the instance was deleted, the request reports back a NOMATCH condition and prompts you for the next transaction.

By testing the FOCURRENT variable, MODIFY requests can process transactions after they have been rejected because of conflicts.

Managing Your Data: Advanced Features

In addition to the basic operations of the MODIFY facility, many other features are available to help you refine your MODIFY requests. This section describes them briefly.

Feature	Description
Absolute File Integrity	Causes FOCUS to write changes to the data source to another section of the disk rather than overwriting the data source. If the request executes normally, the new section of the disk becomes part of the data source. If the system fails, the original data source is preserved.
ACTIVATE	Activates an inactive transaction field. It declares a transaction field to be present so the transaction field can be used for matching, including, and updating. The MOVE option equates the transaction value of the transaction field to the corresponding data source field. The RETAIN option does not move the data source value to the transaction field.
DEACTIVATE (RETAIN)	Deactivates a transaction field. The DEACTIVATE command changes a transaction value to blank if alphanumeric, to zero if numeric, or to the MISSING transaction value for transaction fields described by the MISSING=ON attribute. It also deactivates the corresponding data source field. The RETAIN option deactivates the field without changing its value.
CHECK	Limits the number of transactions lost if the system fails when you are modifying a data source by identifying a checkpoint. CHECK activates the Checkpoint facility that enables FOCUS to write more frequently to the data source. (The point at which the transactions are written is called the "checkpoint.") The Checkpoint Facility is useful in cases when a system failure occurs while MODIFY requests are executing.
COMBINE	Enables you to modify multiple FOCUS data sources in one MODIFY request.
COMMIT and ROLLBACK	Control the changes made to data sources and protect the data sources from system failures. COMMIT and ROLLBACK improve SU performance; here the ability to group individual transactions as one logical transaction reduces the number of individual transactions and the amount of communication needed between the FOCUS Database Server and source users. COMMIT and ROLLBACK are used in lieu of CHECK.

Feature	Description
COMPILE	Translates MODIFY requests into compiled code ready for execution.
COMPUTE	Enables you to modify incoming data field values and to define temporary fields.
DECODE	Enables you to compare transaction values against a list of acceptable and unacceptable values.
LOOKUP	Tests for the existence of non-indexed values in cross-referenced FOCUS data sources and makes these values available for other computations.
ECHO	Displays the logical structure of MODIFY requests. This feature is a good debugging tool for analyzing a MODIFY request, especially if the logic is complex and MATCH and NEXT defaults are used.
FIND	Searches another FOCUS data source for the presence of the transaction value.
LOG	Enables you to record transactions and error messages in separate files automatically, and to control the display of rejection messages at the terminal.
MULTIPLE RECORD PROCESSING COMMANDS	Enable you to process multiple segment instances at one time and are often used with CRTFORM. A few of the important commands used in multiple record processing are GETHOLD and REPEAT. GETHOLD retrieves transaction records from memory and uses them to modify a data source; GETHOLD collects and retrieves segment instances. REPEAT does re-iterative processing.
TYPE	Displays or stores messages in a separate file that you prepare.
VALIDATE	Enables you to reject transactions that contain unacceptable values.

MODIFY Command Syntax

The general syntax of the MODIFY command is

```
MODIFY FILE filename [ECHO|TRACE]
.
.
statements
.
.
DATA [ON ddname|VIA program]
.
incoming data
.
.
[END]
```

where:

MODIFY FILE

Begins the request.

filename

Is the name of the FOCUS data source you are modifying. This name must be the same as the Master File of the data source.

ECHO

Invokes the ECHO facility, which displays the request logic (see *Displaying MODIFY Request Logic: The ECHO Facility* on page 9-209).

TRACE

Invokes the TRACE facility, which displays the name of each case that is entered during the execution of the request if the request uses Case Logic (see *Tracing Case Logic: The TRACE Facility* on page 9-157).

statements

Are the MODIFY statements in the request. Each statement must begin on a separate line.

DATA

Specifies the source of incoming data. Note that nothing should come between this statement and the END statement, unless you are supplying the incoming data in the request itself. In that case, place the data after the DATA statement.

ON *ddname*

Is a DATA statement parameter. See *Specifying the Source of Data: The DATA Statement* on page 9-56.

VIA program

Is a DATA statement parameter.

incoming data

Is the data you are using to modify the data source if you are supplying the data in the request itself.

END

Concludes the request. Do not add this statement if the request contains PROMPT statements (PROMPT statements are discussed in *Prompting for Data One Field at a Time: The PROMPT Statement* on page 9-41).

Executing MODIFY Requests

You can enter and run a MODIFY request either by entering it at the terminal or by running it as a stored procedure (stored procedures are discussed in the *Developing Applications* manual). When you start execution of the request, FOCUS executes the request for each transaction until:

- There is no more data to be read in the incoming transaction data source (the file containing the incoming data).
- The user signals a halt (if the request is prompting the user for data).
- The STOP statement signals a halt to the processing of transactions in an incoming data source (see *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* on page 9-57).
- The request encounters a GOTO EXIT statement.

Syntax

How to Execute a Request as a Stored Procedure

To enter a MODIFY request as a stored procedure, type the request in a procedure file (procedures are discussed in the *Developing Applications* manual). If you are including the incoming data in the request (which you might do for testing purposes), place the data after the DATA statement in the stored procedure. End the request with the END statement unless the request contains PROMPT statements.

After saving the file, enter at the FOCUS prompt

EX focexec

where *focexec* is the name of the stored procedure.

FOCUS responds with an echo of the file name, date, and time as follows:

filename ON date AT time

The request then either begins prompting you for data or starts reading the stored transactions.

When the request finishes execution, it displays the following statistics

```
TRANSACTIONS:  TOTAL   = n  ACCEPTED  = n  REJECTED  = n
SEGMENTS:     INPUT    = n  UPDATED   = n  DELETED   = n
```

where:

n

Is an integer.

TRANSACTIONS

Are the transactions processed by the request.

TOTAL

Is the total number of transactions processed.

ACCEPTED

Is the number of transactions accepted by the request and used to maintain the data source.

REJECTED

Is the number of transactions rejected by the request.

SEGMENTS

Is the number of segment instances modified by the request.

INPUT

Is the number of new segment instances.

UPDATED

Is the number of instances updated.

DELETED

Is the number of instances deleted.

To suppress this message, include the following command in the procedure before the MODIFY request:

```
SET MESSAGE = OFF
```

Syntax **How to Execute MODIFY Requests Online**

To execute a MODIFY request online, enter

```
MODIFY FILE filename
```

where *filename* is the FOCUS name of the data source you are modifying.

FOCUS responds with an echo of the data source name, date, and time as follows:

```
filename ON date AT time  
ENTER SUBCOMMANDS:
```

Enter each MODIFY statement in the request (such as FIXFORM, MATCH, COMPUTE, TYPE) followed by a DATA statement and the incoming data (if the data is not coming from another data source or from the terminal). Then enter the END statement (unless the request contains PROMPT statements).

The request can then start prompting you for data, read from an external data source, or accept transaction records from the terminal (if the request contains FIXFORM or FREEFORM statements but does not specify the ddname of an external data source).

If it accepts transaction records from the terminal, the request appears:

```
START:
```

Start entering the data, one record at a time. Every time you enter a record, the request processes it and displays a message if it rejects the record. After you have entered the data, enter the END statement. This ends execution.

If you are entering a MODIFY request online and you want to cancel the request and start over, enter QUIT. This returns you to the FOCUS prompt.

If you enter a statement online that FOCUS considers an error, it will prompt you for a correction. This error correction facility is described in the *Creating Reports* manual.

You should not enter MODIFY requests online unless the requests are short. If you enter a statement you want to change, you must quit the request and start over.

The example below shows a sample MODIFY request being entered online:

```
>
modify file employee

  EMPLOYEEFOCUS A1 ON 08/15/85 AT 16.36.05
  ENTER SUBCOMMANDS:
  freeform emp_id curr_sal
  match emp_id
  on nomatch reject
  on match update curr_sal
  data
  START:
  emp_id=071382660, curr_sal=21400.50, $
  emp_id=112847612, curr_sal=20350.00, $
  emp_id=117593129, curr_sal=22600.34, $
  end
```

Notice that when the request finishes execution, it displays the following statistics:

```
TRANSACTIONS:  TOTAL= 3  ACCEPTED= 3  REJECTED= 0
SEGMENTS:      INPUT= 0  UPDATED= 3  DELETED= 0
```

These statistics are explained in the preceding section.

Other Ways of Maintaining FOCUS Data Sources

Although the MODIFY command is one of the primary methods of maintaining FOCUS data sources, there are four other facilities for changing data in FOCUS data sources:

- The Maintain facility allows you to maintain data sources (including FOCUS, DB2, SQL/DS, Oracle, Teradata, and VSAM data sources) using event-driven and set-based processing in with a Graphical User Interface. The Maintain facility is described in Chapter 1, *Introduction to Maintain*, through Chapter 8, *Expressions Reference*.
- The FSCAN and SCAN facility allows you to edit FOCUS data sources interactively on a field-by-field basis. You enter a subcommand to make each change. The facility can update key fields. The FSCAN facility is the subject of Chapter 13, *Directly Editing FOCUS Databases With FSCAN*. SCAN is the subject of Chapter 12, *Directly Editing FOCUS Databases With SCAN*.
- The Host Language Interface (HLI) allows you to maintain FOCUS data sources from computer programs written in BAL, FORTRAN, COBOL, and PL/1. HLI is covered in the *Host Language Interface Users Manual*.

Unlike the FSCAN facility mentioned above, the MODIFY command allows you to make many changes with one execution. It can run in both interactive and batch modes. It will prompt you for the values it needs to make the changes, or it may read the values from a transaction data source. However, it cannot update key fields.

Note that although the FOCUS Report Writer can write reports from many kinds of non-FOCUS data sources (such as ISAM, VSAM, and IMS data sources), the MODIFY command maintains only FOCUS data sources, and with the proper interface, VSAM data sources, and SQL and Teradata tables.

You can only MODIFY one partition of a partitioned FOCUS data source at one time. You must explicitly allocate the partition to be modified. Alternatively, you can create separate Master Files for each partition for use in MODIFY procedures. For more information about partitioned FOCUS data sources, see the *Describing Data* manual.

The EMPLOYEE Data Source

The examples in this chapter use the EMPLOYEE data source, a data source used to record employee information for a company. The Master File and the diagram of the entire data source structure are shown in Appendix A, *Master Files and Diagrams*. Most of the examples use three segments in the EMPLOYEE data source:

- The EMPINFO segment contains information directly relating to employees in a company: employee ID, last name, first name, hire date, department assignment, current salary, job code, and classroom hours.
- The SALINFO segment contains information relating to employees' monthly pay: the pay date and the amount of pay.
- The DEDUCT segment contains information about the deductions taken off each monthly pay check: the type of deduction and the amount of the deduction.

Describing Incoming Data

This section describes the statements that read and describe transactions. These are the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements. The last part of the section discusses the DATA, START, and STOP statements.

To modify a data source, the MODIFY request first reads incoming data. It then uses this data to select the segment instances that must be changed or deleted, or to confirm that the instances have not been entered yet and to add them. The data may be in fixed or comma-delimited format, it may be stored in sequential data sources or within the request itself, and it may be entered directly by users on terminals.

There are four MODIFY statements that read and describe incoming data. Some read data from sequential data sources and the request itself; some prompt users on terminals for data. They are:

FIXFORM	Reads data in fixed format. That is, the fields occupy fixed positions in each record.
FREEFORM	Reads data in comma-delimited format. That is, the fields in each record are separated by a comma (,). Each record is terminated by a comma and a dollar sign (,\$).
PROMPT	Prompts users on terminals for data values one field at a time. This statement works on all terminals.
CRTFORM	Displays formatted screens (called CRTFORMs) on terminals and allows users to enter multiple data values at one time.

Note: PROMPT, FREEFORM, FIXFORM, and CRTFORM statements accept data that includes numbers expressed in scientific notation. For more information on the use of scientific notation in expressions, refer to the *Creating Reports* manual.

If a request does not have one of these statements, it defaults to FREEFORM and reads data from a comma-delimited list.

These statements can be placed in requests in two ways:

- The statements can stand by themselves. These statements read data every time the request repeats.
- The statements can be phrases in MATCH or NEXT statements (discussed in *Modifying Data: MATCH and NEXT* on page 9-58). These phrases only read data when the MATCH or NEXT statement is executed.

A request may have an unlimited number of statements of one type (for example, 10 PROMPT statements), except for CRTFORM where up to 255 such statements are allowed. You may also mix the following statements in one request:

- FREEFORM statements and PROMPT statements.
- One FIXFORM statement with up to 255 CRTFORMs.

If you are reading data from a data source or user program, you must allocate the source of the data to a ddname.

Note: Do not begin any field used in a CRTFORM or FIXFORM statement with Xn , where n is any numeric value. This applies to fields in the Master File and computed fields.

FOCUS allows the use of up to 3,072 fields in each MODIFY request. This total includes both data source fields and temporary fields.

The last part of the section discusses several other features related to reading transactions. They are:

- The DATA statement that marks the end of the executable portion of the request and specifies the source of the transactions (the request itself, a data source, the terminal, or a user program).
- The START and STOP statements that limit the request to reading a portion of the transaction data source.

Reading Fixed-Format Data: The FIXFORM Statement

The FIXFORM statement reads data in fixed format. That is, each field has a fixed position in each record. The FIXFORM statement can read data from sequential data sources, including HOLD, SAVE, and SAVB files generated by TABLE requests.

The FIXFORM statement reads in one logical record at a time starting from column one and divides the record into transaction fields. Subsequent FIXFORM statements may redefine the record, dividing it into different sets of fields.

Note: Multiple FIXFORM statements in a request can function as a single statement.

For example, you are adding the names of five new employees to the EMPLOYEE data source. The data is stored in a sequential data source called NEWEMP.

This is how the data source appears on a text editor such as TED:

```
|....+....1....+....2....+....3....+....4
* * * TOP OF FILE * * *
222333444BLACK      SUSAN      27500.00
456456456NEWMAN    JERRY      24800.75
999888777HUNTINGTON LAWRENCE  26950.00
246246246LINDQUIST DEBRA      19300.40
666888222MCINTYRE  GEORGE     31900.60
* * * END OF FILE * * *
```

Each record in the data source consists of four fields, each field in a fixed position on the record:

- The EMP_ID field (employee ID numbers) occupies the first nine bytes of each record (columns 1 through 9).
- The LAST_NAME field occupies the next ten bytes (columns 10 through 19).
- The FIRST_NAME field occupies the next ten bytes (columns 20 through 29).
- The CURR_SAL field (current salaries) occupies the last eight bytes in each record (columns 30 through 37).

You can describe the record format with this FIXFORM statement:

```
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/10 CURR_SAL/8
```

To add the records to the FOCUS data source, include the preceding statement in this MODIFY request:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/10 CURR_SAL/8

MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA ON NEWEMP
END
```

Syntax **How to Use a FIXFORM Statement**

The syntax of the FIXFORM statement is

```
FIXFORM [ON ddname] fld-1/form-1 ... fld-n/form-n
```

or

```
FIXFORM FROM master [ALIAS]
```

where:

fld-1 ...

Are the names of the incoming data fields that the FIXFORM statement is reading or redefining. If the name has an embedded blank, enclose it within single quotation marks.

Any field being read by the FIXFORM statement that does not appear in the Master File of the data source being modified must be predefined in a COMPUTE field/format=; statement. This COMPUTE must appear in the MODIFY before the FIXFORM.

The list of fields must fit on one line. If the list is too long to fit on one line, use a FIXFORM statement for each line. For example:

```
FIXFORM EMP_ID/9 LAST_NAME/15  
FIXFORM CURR_SAL/8 ED_HRS/4
```

The two FIXFORM statements act as one statement and read one record into the buffer.

form-1 ...

Are the formats of the incoming data fields, as described in *How to Specify Field Formats With FIXFORM* on page 9-26. The formats specify the format type (alphanumeric, integer, floating point, and so on) and the length of the field in bytes.

Note: No length is specified for the text field format that is variable in length. A FIXFORM statement can describe up to 12,288 bytes, exclusive of repeating values.

To specify an alphanumeric format, type the length of the field in bytes. For example, a record contains two alphanumeric fields:

The EMP_ID field, nine bytes long.

The DEPARTMENT field, ten bytes long.

The FIXFORM statement that describes this record is:

```
FIXFORM EMP_ID/9 DEPARTMENT/10
```

Note that alphanumeric transaction fields can modify any data source field regardless of internal format. Specifying the formats of binary, packed, and zoned transaction fields is discussed in *How to Specify Field Formats With FIXFORM* on page 9-26.

Remember that a transaction field can contain numbers and still be alphanumeric. If you display a transaction data source on a system editor, alphanumeric data appears normally; numeric data appears as unprintable hexadecimal characters.

ON *ddname*

Is an option that specifies the *ddname* of the transaction data source containing the incoming data. You use this option most often when the request is reading data from two different sources: one source is specified by the DATA statement, the other by the ON *ddname* option.

Note that if there is more than one FIXFORM statement without the ON *ddname* option, the request keeps track of the last column of the physical record read by the last FIXFORM statement. If the last column is in the middle of the record, the next FIXFORM statement begins to read from the next column. If the last column is at the end of the record, the next FIXFORM statement begins to read from column 1 of the next record.

To break a FIXFORM statement having the ON *ddname* option into smaller statements, specify the ON *ddname* option only in the first statement. All the statements must be together in one block. For example:

```
FIXFORM ON EMPFILE EMP_ID/9 LAST_NAME/15
FIXFORM FIRST_NAME/10 DEPARTMENT/10
FIXFORM CURR_SAL/8 ED_HRS/4
```

FROM *master*

Indicates that the incoming data fields have the same names and formats as the Master File (named *master*). If you use this option, do not specify the field names and formats in the FIXFORM statement itself. Use this option only if the Master File specifies a single segment SUFFIX=FIX data source. All the fields in the Master File specified by the FROM phrase must also appear in the Master File specified by the MODIFY command, or an error will result.

You use this option most often to load data from a HOLD file. For example:

```
TABLE FILE EMPLOYEE
PRINT CURR_SAL BY EMP_ID
ON TABLE HOLD
END
MODIFY FILE SALARY
FIXFORM FROM HOLD
DATA ON HOLD
END
```

The TABLE request stores employee IDs and salaries in a HOLD file. The MODIFY request loads the IDs and salaries into a new FOCUS data source called SALARY. Note that all the fields in the HOLD Master File must also appear in the SALARY Master File.

Text fields are supported with FIXFORM from HOLD; only one text field can be read from a HOLD file and it must be the last field on the HOLD FIXFORM. The representation of missing text depends on whether MISSING=ON in the Master File or the FIXFORM format is C for conditional, or a combination of the two.

When duplicate field names exist in a HOLD file, a MODIFY request that includes FIXFORM FROM HOLD should specify an AS name.

Note: FIXFORM FROM Master File automatically assumes that all fields on the FIXFORM are conditional fields. Because of this a value of blank does not update the database to a value of blank. If blank (or spaces) is a valid value, and the update should take place, you must issue an ACTIVATE RETAIN fieldname fieldname fieldname... or ACTIVATE RETAIN SEG.fieldname.

ALIAS

Indicates that the alias names from the Master File are to be used to build the FIXFORM statements.

Syntax How to Skip Columns in the Record

Often, an incoming transaction contains filler or data you do not need. To skip over characters or information in the incoming record, type

`Xn`

where:

`n`

Is the number of columns you want to skip.

This does not cause the statement to ignore the skipped columns. The statement reads the entire record; it just does not place the skipped data in any transaction field. Later in the request, you can place this data into transaction fields by adding a second FIXFORM statement (see the following section, *Moving Backward Through a Record* on page 9-25).

For example, a transaction record consists of two fields: EMP_ID and CURR_SAL. Two "A"s separate the fields:

```
071382660AA23540.35
```

You describe this record with this FIXFORM statement:

```
FIXFORM EMP_ID/9 X2 CURR_SAL/8
```

The X2 notation prevents the two "A"s from being placed in the transaction fields.

Note: Do not begin any field used in a CRTFORM or FIXFORM statement with `Xn`, where `n` is any numeric value. This applies to fields in the Master File and computed fields.

Procedure Moving Backward Through a Record

After a FIXFORM statement reads a record into the buffer, it places the data into transaction fields, starting from the beginning of the record and moving toward the end. You can specify that FIXFORM back up a number of columns to process the data more than once. This enables you to place the same data into two fields simultaneously. To do this, use the notation

`X-n`

where *n* is the number of columns that the statement is to move backward. For example, the first three digits of employee IDs are a special code that you wish to use later in the request. Each employee ID is nine digits long. You type this FIXFORM statement:

```
FIXFORM EMP_ID/9 X-9 EMP_CODE/3 X6 CURR_SAL/8
```

A record in the transaction data source is:

```
07138266023500.35
```

The statement interprets the record this way:

<code>EMP_ID/9</code>	Reads the first nine bytes as the employee ID (071382660).
<code>X-9</code>	Goes back nine bytes to the beginning of the record.
<code>EMP_CODE/3</code>	Reads the first three bytes as the employee code (071).
<code>X6</code>	Moves forward six bytes.
<code>CURR_SAL/8</code>	Reads the next eight bytes as the employee salary (23500.35).

This defines three incoming fields, all of which you can use later in the request.

Note: Since the EMP_CODE field is not defined in the Master File, you must define the field with the COMPUTE statement before the FIXFORM statement (see *Computing Values: The COMPUTE Statement* on page 9-93).

You may replace any FIXFORM statement with two smaller statements so that the second statement redefines all or part of the record read by the first statement. For example, you may replace this FIXFORM statement

```
FIXFORM EMP_ID/9 X-9 EMP_CODE/3 X6 CURR_SAL/8
```

with these two smaller FIXFORM statements:

```
FIXFORM EMP_ID/9 CURR_SAL/8
FIXFORM X-17 EMP_CODE/3 X14
```

The first FIXFORM statement reads one record and divides the record into the EMP_ID field (nine bytes) and the CURR_SAL field (eight bytes).

The second FIXFORM statement moves 17 bytes back to the beginning of the record and declares the first three bytes to be the EMP_CODE field. It then skips over the last 14 bytes.

Note that you cannot place the *X-n* notation at the end of a FIXFORM statement. The following statement is an error:

```
FIXFORM EMP_ID/9 CURR_SAL/8 X-17
```

FIXFORM statements that redefine records in the buffer are especially useful in Case Logic requests (see *Case Logic Applications* on page 9-148).

Syntax **How to Specify Field Formats With FIXFORM**

This section lists the data formats that may be specified in FIXFORM statements. In addition to alphanumeric format, there are date (DATE), text field (TX), and conditional text field (CTX) formats, and numeric formats of fields in HOLD and SAVB files and of fields generated by user-written programs. The formats are

```
[A] n [YQMD]   In [YQMD]   F4   D8   Pn [.m] [YQMD]   DATE   /TX   /CTX   Zn [.m]
```

where:

[A] n [YQMD]

Specifies an alphanumeric character string *n* bytes long, where *n* is an integer. Date component options (YY, Y, Q, M, D) are included as necessary for a date field.

In [YQMD]

Specifies a binary integer *n* bytes long, where *n* is 1, 2, or 4. Date component options (YY, Y, Q, M, D) are included as necessary for a date field.

F4

Specifies a 4-byte binary floating point number.

D8

Specifies an 8-byte binary double precision number.

Pn [.m] [YQMD]

Specifies a packed number *n* bytes long with *m* digits after an implied decimal point. *n* is an integer between 1 and 16 and *m* is an integer between 0 and 33. Date component options (YY, Y, Q, M, D) are included as necessary for a date field.

DATE

Specifies a date field in 4-byte integer format, to be copied to the data source without date translation or validation. Date format fields can also be read without these restrictions by specifying alphanumeric, integer, or packed format, as described later in this section.

`/TX|/CX`

Specifies a text field format for transaction and conditional transaction fields. Each FIXFORM statement can include one of these fields, which must appear as the last field in the statement. Note that you do not specify the length when using FIXFORM to read text fields; the length is for display purposes only (see the *Describing Data* manual). See *Entering Text Data Using TED* on page 9-52 for general rules.

Note:

- If more than one text field exists in the Master File, you must read each one using a separate FIXFORM statement. The text field must be the last field listed in the FIXFORM statement.
- If the word END appears on a line by itself, FOCUS interprets it as a quit action, stops the procedure, and discards everything entered up to that point for a particular record.
- To end a transaction and exit MODIFY, first enter the end-of-text character (%\$) on a line by itself, then enter END on the next line.
- If data is read from an external data source, the record format must be fixed.
- If the text field is not mentioned in the FIXFORM statement, but it is present in the Master File, the value of the text field is determined based on the setting of the MISSING attribute. That is, if MISSING=ON, the text will be entered as a dot (.). If MISSING=OFF, the text will be entered as a blank.

`Zn [. m]`

Specifies a zoned decimal number n bytes long with m digits after an implied decimal point. n is an integer between 1 and 16 and m is an integer between 0 and 9.

For example, this FIXFORM statement

```
FIXFORM EMP_ID/9 HIRE_DATE/I4 CURR_SAL/D8 ED_HRS/P4.2
```

defines each record as the following:

- The first nine bytes as the character string EMP_ID.
- The next four bytes as the binary integer HIRE_DATE.
- The next eight bytes as the binary double precision number CURR_SAL.
- The next four bytes as the packed number ED_HRS. The last two digits of the number follow an implied decimal point.

The FIXFORM statement specifies the field formats of transaction data sources, not the data source being updated. A transaction field can modify a data source field if the transaction field has one of the following format types (the format type is the type of field, such as alphanumeric or floating point):

- The same format type as the data source field.
- Alphanumeric format.
- Zoned format (if the data source field is packed).

If you specify any other format type for the transaction field (for example, an integer transaction field to modify a floating point data source field), the request may terminate and generate an error message. To read such a transaction value into a data source field, do the following:

1. Before the FIXFORM statement, use the COMPUTE statement to define a name for the incoming data field that is different from the data source field (the COMPUTE statement is discussed in *Computations: COMPUTE and VALIDATE* on page 9-92). The statement also specifies the field format, showing the format type and the number of digits in the field.
2. In the FIXFORM statement, read the incoming data field using the name you defined in the COMPUTE statement. The field format in the FIXFORM statement shows the field length in bytes in the transaction data source.
3. After the FIXFORM statement, use the COMPUTE statement to set a field with the same name as the data source field equal to the value of the field you defined in step 1.

Note: If the incoming field is numeric and the data source field is alphanumeric, use the EDIT function to do this. The EDIT function is described in the *Creating Reports* manual.

The following request reads a floating point field called FLOATSAL into the data source double-precision field CURR_SAL:

```
MODIFY FILE EMPLOYEE
COMPUTE FLOATSAL/F8=;
FIXFORM EMP_ID/12 FLOATSAL/F4
COMPUTE CURR_SAL = FLOATSAL;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA ON FLOAFILE
END
```

Notice that the FLOATSAL field is defined with a format of F8 in the first COMPUTE statement and a format of F4 in the FIXFORM statement. FLOATSAL is an eight-digit field that takes up four bytes in the transaction data source.

Describing Date Fields

This section discusses using date format fields in FIXFORM statements. Alphanumeric and integer format fields with date edit options are not discussed here; they are treated by FIXFORM like standard alphanumeric and integer fields.

When you use a FIXFORM statement to modify a data source date field, the corresponding data in the transaction data source can be one of the following three types:

- A numeric date literal. For example, August 17 1989 can be represented in the transaction data source as 081789. The transaction field format can be *An*, *In*, or *Pn*.
- A natural date literal. For example, August 17 1989 can be represented in the transaction data source as AUG 17 1989. The transaction field format must be *An*.

Note that all names of days and months in the transaction data source must be in uppercase, even if the translation option is *t* or *tr*. All abbreviated names of days and months in the transaction data source must consist of the first three letters of the name. Commas cannot be included in the date.

- A date in internal FOCUS date format. This format is used for date fields in SAVB and unformatted HOLD files. The date is stored as a 4-byte integer representing the elapsed time since the standard FOCUS base date, as described in the *Describing Data* manual. The transaction field format must be DATE.

For example, assume that you have changed the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FIXFORM statement:

```
FIXFORM EMP_ID/11 FIRST_NAME/10 LAST_NAME/10 HIRE_DATE/9
```

Both of these records are valid input:

```
444555666 DOROTHY TAILOR 860613
444555666 DOROTHY TAILOR 86 JUN 13
```

To describe date fields in FIXFORM statements, you can use the following transaction field formats.

- **DATE.** This specifies a transaction field stored in FOCUS internal date format, which is a 4-byte integer representing the time elapsed from the standard FOCUS base date, as described in the *Describing Data* manual. The transaction field will be copied directly to the data source without date validation.

For example:

```
FIXFORM SALEDATE/DATE
```

- **An, In, Pn.** These specify a date field stored in alphanumeric, integer, or packed decimal format respectively. Numeric date literals and natural date literals are translated as necessary to suit the data source field's USAGE specification and edit options.

For example, if a data source contains the date field NEWSDATE, and USAGE=MDYY, the following FIXFORM statements can be used to update NEWSDATE:

```
FIXFORM NEWSDATE/A8YYMD  
FIXFORM NEWSDATE/A6DMY  
FIXFORM NEWSDATE/I4MDY  
FIXFORM NEWSDATE/I2YMD  
FIXFORM NEWSDATE/P3DMY  
FIXFORM NEWSDATE/A8
```

Note that the last FIXFORM statement does not specify any date components. Because it is alphanumeric and has the same length specified by the data source field's USAGE attribute, it defaults to the USAGE format (which in this case is MDYY).

For all date transaction field formats, the date components (year, quarter, month, day) do not need to be in the order specified in the USAGE attribute in the Master File; they can be in any order.

Note, however, that you cannot extract date components from a date field (for example, you cannot write a YMD transaction field to a YM data source field), and you cannot convert one component to another (for example, you cannot convert a YM transaction field to a YQ data source field). The only exceptions are the YY and Y date components, which can be substituted for each other.

Syntax **How to Describe Repeating Groups**

You may use a fixed-format transaction record to modify multiple segment instances. The set of transaction fields that modify the instances is called a repeating group because the fields repeat for each instance. Instead of explicitly specifying each field, you specify the repeating group once with a multiplying factor in front.

The syntax is

```
FIXFORM factor (group)
```

where:

factor

Is the number of times that the group repeats.

group

Is the repeating group consisting of a list of fields and formats.

For example, assume you design a request that records the last 12 months of employees' monthly pay in the EMPLOYEE data source. Each transaction record contains the employee's ID and 12 pairs of fields: the first field in each pair is the pay date, the second is the monthly pay (GROSS). The request is:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 12 (PAY_DATE/6 GROSS/7)
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA ON EMPGROSS
END
```

Each incoming record that the request reads contains one EMP_ID field and 12 groups of fields, each group consisting of a pay date field and a monthly pay field. The request reads a record, then splits the record into 12 smaller logical records, each consisting of the employee ID of the original record and one group. FOCUS then executes the request for each logical record, processing each group separately.

You may specify more than one group in a FIXFORM statement, but they cannot be nested.

Note: To process repeating groups in a Case Logic request, place each repeating group in a FIXFORM statement in a separate case. The case should include the following:

- A counter that counts the group being processed.
- An IF statement that branches out of the case after all the groups are processed.
- GOTO phrases that branch back to the beginning of the case after each group is processed.

The following request adds and updates information on employees' monthly pay. Note the ON INVALID phrase that branches back to the beginning of the case if a monthly pay entry is greater than \$2500. The request is:

```
MODIFY FILE EMPLOYEE
COMPUTE
  COUNTER/I3 = 0;
FIXFORM EMP_ID/9
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO NEWPAY
GOTO NEWPAY

CASE NEWPAY
COMPUTE
  COUNTER/I1 = COUNTER + 1;
IF COUNTER GT 3 GOTO TOP;
FIXFORM 3 (PAY_DATE/6 GROSS/7)
VALIDATE
  PAYTEST = IF GROSS GT 2500 THEN 0 ELSE 1;
  ON INVALID GOTO NEWPAY
MATCH PAY_DATE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO NEWPAY
  ON MATCH UPDATE GROSS
  ON MATCH GOTO NEWPAY
ENDCASE
DATA ON PAYFILE
END
```

Example Conditional Fields

MODIFY requests can process records in which alphanumeric field values may be present in one input record but absent in another. Such fields are called conditional fields. When the value of a conditional field is blank, the request does not use the field to modify the data source and the field remains inactive (active and inactive fields are discussed in *Active and Inactive Fields* on page 9-199).

To indicate to FOCUS that a field is conditional, precede the field format with the letter C. For example:

```
FIXFORM FIRST_NAME/C10 LAST_NAME/C15
```

Another example: You design a MODIFY request that updates employees' departments and job codes. If an employee's department or job code has not changed, the corresponding field in the transaction data source is blank.

The request is:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 DEPARTMENT/C10 X1 CURR_JOBCODE/C3
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE DEPARTMENT CURR_JOBCODE
DATA
071382660 SALES    B13
112847612 A08
117593129 MARKETING
END
```

The request contains three incoming records after the DATA statement:

- The first incoming record contains all three fields. The request updates both the DEPARTMENT and CURR_JOBCODE fields.
- The next record has the EMP_ID and CURR_JOBCODE fields but no DEPARTMENT field. The request updates the employee's CURR_JOBCODE value in the data source, but leaves the DEPARTMENT value the same.
- The last record has the EMP_ID and DEPARTMENT fields but no CURR_JOBCODE field. The request updates the employee's DEPARTMENT value in the data source, but leaves the CURR_JOBCODE value the same.

If you did not describe the DEPARTMENT and CURR_JOBCODE fields as conditional, the request would change an employee's department or job code to blank whenever these fields in the incoming records were blank.

If you are adding segment instances, and several fields are conditional, values that are blank go into the new instances as:

- Blank, if the instance fields are alphanumeric.
- Zero, if the instance fields are numeric.
- The MISSING symbol, if the fields are described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

Example **FIXFORM Phrases in MATCH and NEXT Statements**

You may use FIXFORM statements as phrases in MATCH and NEXT statements. These phrases are useful if you want to read records selectively only if a particular segment instance exists in the data source (or is confirmed not to be in the data source).

For example, you design a MODIFY request that adds records of employees' monthly pay to the data source:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 PAY_DATE/6
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE

MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH FIXFORM ON MONTHPAY GROSS/7
  ON NOMATCH INCLUDE

DATA ON EMPPAY
END
```

The data is kept in two transaction data sources: EMPPAY and MONTHPAY. The EMPPAY data source contains the employee IDs and the date each employee was paid. The MONTHPAY data source contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

- The employee ID is recorded in the data source. This is confirmed by the MATCH EMP_ID statement.
- The date the employee was paid has not yet been recorded in the data source. This is confirmed by the MATCH PAY_DATE statement.

Once the request has confirmed this, it can read the monthly pay from the MONTHPAY data source

```
ON NOMATCH FIXFORM ON MONTHPAY GROSS/7
```

and record it in the data source:

```
ON NOMATCH INCLUDE
```

Reading in Comma-delimited Data: The FREEFORM Statement

The FREEFORM statement reads comma-delimited data, where field values in each record are separated by commas, and records are terminated by comma-dollar signs (,\$). The data may be stored in the request itself or in separate sequential data sources.

If the MODIFY request does not provide a statement reading transactions (FIXFORM, FREEFORM, PROMPT, or CRTFORM), FREEFORM is the default.

The following request updates employee salaries by reading employee IDs and new salaries from comma-delimited records. The records follow the DATA statement:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
EMP_ID=071382660, CURR_SAL=21400.50, $
EMP_ID=112847612, CURR_SAL=20350.00, $
EMP_ID=117593129, CURR_SAL=22600.34, $
END
```

Syntax **How to Use a FREEFORM Statement**

The syntax of the FREEFORM statement is

```
FREEFORM [ON ddname] [field-1 field-2 ... field-n]
```

where:

ON ddname

Is an option that specifies the *ddname* of the transaction data source containing the incoming data. Use this option only when the DATA statement does not specify a *ddname* or specifies a *ddname* of a different data source.

field-1 ...

Are the names of the fields in the order that they appear in the record.

Note: FREEFORM follows the same rules as FIXFORM when dealing with TEXT fields. For more information see *Reading Fixed-Format Data: The FIXFORM Statement* on page 9-20.

If the order of fields is specified in the data, you do not need it in the syntax and if the order of fields is specified in the syntax, you do not need it in the data.

The list of fields must fit on one line. If the list is too long for a single line, use a FREEFORM statement for each line. For example:

```
FREEFORM EMP_ID LAST_NAME FIRST_NAME  
FREEFORM DEPARTMENT CURR_SAL
```

These two FREEFORM statements act as one statement and read one record into the buffer.

Each time a FREEFORM statement is executed, it reads one record up to the comma-dollar sign (,\$). It does not read beyond that. If the FREEFORM command is used with incoming data having embedded commas, the data must be enclosed in single quotation marks in the input data source.

If a MODIFY request has a FREEFORM statement, the statement must specify all the fields in the transaction data source. If the transaction data source has fields not specified in the FREEFORM statement, the request terminates and generates an error message.

If you do not include a transaction statement in your MODIFY request, the request assumes the default FREEFORM and expects to read comma-delimited data. The request reads one record every time it executes the first statement in the request. Nevertheless, you should include a FREEFORM statement to make clear that the request is reading comma-delimited data, to show when the request reads the data, and to allow greater flexibility in entering data into comma-delimited data sources.

If the Master File lists a date format with a translation option (see the *Describing Data* manual), you can type the date values in the transaction data source as they appear in reports generated by TABLE requests (but do not type the commas in the dates). Note the following conditions:

- The date format must have had the translation option before the FOCUS data source was created.
- All names of months must be in uppercase, even if the translation option is *t* or *tr*.

For example, assume you change the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FREEFORM statement:

```
FREEFORM EMP_ID FIRST_NAME LAST_NAME HIRE_DATE/9
```

Both these records are valid input:

```
444555666, DOROTHY, TAILOR, 860613, $
444555666, DOROTHY, TAILOR, 86 JUN 13, $
```

Identifying Values in a Comma-delimited Data Source

This section discusses how MODIFY requests identify the values in comma-delimited data sources and determine what fields they belong to. (For more information on comma-delimited data sources, see the *Describing Data* manual.) There are two types of values in comma-delimited data sources:

- Identified values are identified explicitly in the data source.
- Positional values exist by themselves without any identification.

Identified values have the form

```
identifier = value
```

where *identifier* identifies the field to which the value belongs.

Identifiers can be one of two types:

- Field names or unique truncations of field names. For example:

```
DEPARTMENT=SALES, CURR_SAL=25000, $
```

- Aliases. For example:

```
DPT=SALES, CSAL=25000, $
```

If the request has a FREEFORM statement, the statement must specify all identified fields. However, the request identifies the values by their identifiers, not by the order of field names in the FREEFORM list.

Positional values exist by themselves without any identification in the data source. For example:

```
SALES, 25000, $
```

The MODIFY request identifies positional values by the order of field names specified in the FREEFORM statement list. If a record consists only of positional values, the request assigns the first field name in the list to the first value, the second field name in the list to the second value, and so on. For example, if a request has the statement:

```
FREEFORM EMP_ID DEPARTMENT CURR_SAL
```

Then the record

```
071382660, SALES, 25000, $
```

is interpreted this way:

```
EMP_ID: 071382660  
DEPARTMENT: SALES  
CURR_SAL: 25000
```

If a record has both identified and positional values, the MODIFY request identifies the positional values in the following way: it notes the last explicitly identified value to precede the positional values in the record. It then identifies the positional values by the order of field names that follow the name of the explicitly identified field in the FREEFORM list.

For example, a MODIFY request has this FREEFORM statement:

```
FREEFORM EMP_ID FIRST_NAME LAST_NAME CURR_SAL
```

The transaction data source contains this record:

```
FIRST_NAME=DAVID, MCHENRY, 21300.45, $
```

The first value, DAVID, is explicitly identified as the FIRST_NAME field. The request identifies the next value, MCHENRY, as the LAST_NAME field because LAST_NAME follows FIRST_NAME on the FREEFORM list. Similarly, the request identifies 21300.45 as the CURR_SAL field. The EMP_ID field retains the value it was last given.

If the MODIFY request has no FREEFORM statement, it identifies positional values by the order of field names declared in the Master File. If a record consists of only positional values, the request assigns the first field name in the Master File to the first value, the second field name to the second value, and so on. For example, a transaction data source contains this record:

```
071382660, MCHENRY, DAVID, $
```

The request identifies the first value, 071382660, as the EMP_ID field because EMP_ID is the first field in the Master File. The next value, MCHENRY, is the LAST_NAME field (the second field in the Master File). DAVID becomes the FIRST_NAME field, the third field in the Master File (the EMPLOYEE Master File is shown in Appendix A, *Master Files and Diagrams*).

If a record has both identified values and positional values, the MODIFY request identifies the positional values the following way: it notes the last explicitly identified value to precede the positional values in the record. It then identifies the positional values by the order of field names that follow the name of the explicitly identified field in the Master File. For example, the transaction data source contains this record:

```
FIRST_NAME=DAVID, 820406, PRODUCTION, $
```

The first value, DAVID, is explicitly identified as the FIRST_NAME field. The request identifies the next value, 820406, as the HIRE_DATE field because HIRE_DATE follows FIRST_NAME in the Master File. Similarly, the request identifies PRODUCTION as the DEPARTMENT field.

Example Missing Values in Comma-delimited Data Sources

If a field value is missing for a particular record, you must explicitly identify the name of the next field in the record. For instance, a FREEFORM statement specifies the following:

```
FREEFORM EMP_ID CURR_SAL DEPARTMENT
```

One record lacks a CURR_SAL value. Type the record this way

```
071382660, DEPARTMENT=PRODUCTION, $
```

where 071382660 is an EMP_ID value. The CURR_SAL field remains inactive and will not change any CURR_SAL values in the data source.

If you are adding segment instances to the data source, the instance fields not receiving a value become:

- Blank, if the instance fields are alphanumeric.
- Zero, if the instance fields are numeric.
- The MISSING symbol, if the fields are described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

An important exception: If you omit fields from the beginning of a record, the fields retain the values last assigned to them from a previous record. For example, a transaction data source contains these two records:

```
EMP_ID=071382660, PAY_DATE=820831, GROSS=1045.60, $
PAY_DATE=820831, GROSS=1047.20, $
```

The second record is lacking an EMP_ID value. Nevertheless, since EMP_ID is at the beginning of the record, it retains its value of 071382660 for the second record and remains active.

If you use double commas to mark an absent value, the value becomes a blank character string if alphanumeric, and zero if numeric. Note that the request can use this value to modify the data source. For example, in the record

```
071382660,, PRODUCTION, $
```

the two commas mark the position of the absent CURR_SAL field. The CURR_SAL field becomes active and can change an employee salary to \$0.00.

Example **FREEFORM Phrases in MATCH and NEXT Statements**

You may use FREEFORM statements as phrases in MATCH and NEXT statements. These phrases are useful if you want to read records selectively if a particular segment instance exists in the data source (or is confirmed not to be in the data source).

For example, the following MODIFY request adds records of employees' monthly pay to the data source:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID PAY_DATE
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH FREEFORM ON MONTHPAY GROSS
  ON NOMATCH INCLUDE
DATA ON EMPPAY
END
```

The data is kept in two transaction data sources: EMPPAY and MONTHPAY. The EMPPAY data source contains the employee IDs and the date each employee was paid. The MONTHPAY data source contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

- The employee ID is recorded in the data source. This is confirmed by the MATCH EMP_ID statement.
- The date the employee was paid has not yet been recorded in the data source. This is confirmed by the MATCH PAY_DATE statement.

Once the request has confirmed this, it can read the monthly pay from the MONTHPAY data source

```
ON NOMATCH FREEFORM ON MONTHPAY GROSS
```

and record it in the data source:

```
ON NOMATCH INCLUDE
```

Prompting for Data One Field at a Time: The PROMPT Statement

The PROMPT statement prompts the user on a terminal for incoming data one field at a time. Use this statement for requests that may be run on line terminals or by users having no access to the FIDEL facility. If the requests will be run exclusively by users on full-screen terminals with access to FIDEL, use the CRTFORM statement instead. The FIDEL facility and the CRTFORM statement are the subjects of Chapter 10, *Designing Screens With FIDEL*.

Syntax

How to Use a PROMPT Statement

The syntax of the PROMPT statement is

```
PROMPT {field-1[.text.] field-2[.text.] ... field-n[.text.]}*
```

where:

field-1 ...

Are the names of the fields for which you are prompting. An asterisk * instead of field names prompts for all fields described in the Master File in the order that they are declared.

The list of fields must fit on one line. If the list is too long to fit on one line, use a PROMPT statement for each line. For example:

```
PROMPT EMP_ID LAST_NAME FIRST_NAME
PROMPT DEPARTMENT CURR_SAL
```

Each field in the Master File with a text field format must appear in a separate PROMPT statement as the last field in the statement. When prompted for text, note that the length of the text entry is limited only by the amount of virtual storage space. The last line of text data that you enter must be followed by the end-of-text mark (%\$) on a line by itself. For additional guidelines regarding fields with a text field format, see *Entering Text Data Using TED* on page 9-52.

text

Is optional prompting text, up to 38 characters per field.

Do not place an END statement at the end of the request. Conclude the request with the DATA statement.

The following request updates information about employees' department assignments, salaries, and job codes:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT CURR_SAL CURR_JOBCODE
MATCH EMP_ID
  ON MATCH UPDATE DEPARTMENT CURR_SAL CURR_JOBCODE
  ON NOMATCH REJECT
DATA
```

When you execute the command, the following appears on your screen

```
> EMPLOYEE ON 06/19/98 AT 14.38.27  
DATA FOR TRANSACTION 1
```

```
MP_ID= >
```

where:

```
EMPLOYEE
```

Is the system name of the data source (in this case, the TSO name).

```
ON 06/19/98 AT 14.38.27
```

Is the date and time that FOCUS opened the data source: June 19, 1998 at 2:38:27 p.m.

```
DATA FOR TRANSACTION 1
```

Notifies the user that the request is prompting for the first transaction. Each cycle of prompts constitutes one transaction. When the next transaction begins, the request prompts again for the first field in the cycle. In this request, the EMP_ID, DEPARTMENT, CURR_SAL, and CURR_JOBCODE prompts constitute one transaction. When the next transaction begins, the request prompts for the EMP_ID field again.

```
EMP_ID = >
```

Is the default prompt for the EMP_ID field (the field name).

As each prompt appears, enter the value for the field requested. When you finish entering values, end execution by entering End or Quit at any prompt. The following is a sample execution of the request shown above (user input is shown in lowercase; computer responses are in uppercase):

```
> EMPLOYEE ON 06/19/98 AT 14.38.27  
DATA FOR TRANSACTION 1
```

```
EMP_ID      = > 071382660  
DEPARTMENT  = > mis  
CURR_SAL    = > 22500.35  
CURR_JOBCODE = > b12  
DATA FOR TRANSACTION 2
```

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

When you design a request that prompts for fields and validates them, we recommend that validating the field values after every prompt is recommended. This saves extra typing if one of the field values proves invalid. Validation tests are discussed in *Validating Transaction Values: The VALIDATE Statement* on page 9-99.

If the Master File lists a date format with a translation option (see the *Describing Data* manual), you may type the date as it appears in reports generated by TABLE requests (but do not type the commas in the dates). Note that the date format must have had the translation option before the FOCUS data source was created.

For example, assume you change the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FIXFORM statement:

```
PROMPT EMP_ID FIRST_NAME LAST_NAME HIRE_DATE
```

When you execute the request, a sample transaction might appear like this:

```
DATA FOR TRANSACTION 2

EMP_ID           = > 444555666
FIRST_NAME       = > dorothy
LAST_NAME        = > tailor
HIRE_DATE (YMDT) = > 98 jun 13
```

Note that you can also respond to the HIRE_DATE prompt with the value 980613.

Syntax

How to Prompt for Repeating Groups

You may prompt for the same group of fields repeatedly. This is convenient when you want to modify a child segment chain. You prompt once for the key field of the parent instance and prompt repeatedly for the values of the child instances. Without repeating groups, you must prompt for the key field of the parent instance each time you prompt for a child instance.

For example, a MODIFY request updates employees' monthly pay. It first prompts for an employee ID, then for 12 pairs of fields: the first field in each pair is a pay date, the second field is the updated pay. The pay date and updated pay fields are a repeating group.

To specify a repeating group, use the following syntax

```
PROMPT factor (group)
```

where:

factor

Is the number of times the group repeats.

group

Is the repeating group of fields.

Note that the transaction counter that appears during prompting counts each repeating group cycle of prompts as one transaction.

For example, the following request adds three instances of monthly pay (GROSS) for each employee:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID 3 (PAY_DATE GROSS)
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

This request prompts you for an employee ID, then a pay date, a monthly pay, a pay date, a monthly pay, and so on until it prompts you for three pay dates and three monthly pays. It then prompts you for the next employee ID.

The following is a sample execution of the previous request:

```
> EMPLOYEE ON 09/19/98 AT 15.01.38
  DATA FOR TRANSACTION  1

  EMP_ID      = > 071382660
  PAY_DATE    = > 860131
  GROSS       = > 1360.50
  DATA FOR TRANSACTION  2

  PAY_DATE    = > 860228
  GROSS       = > 1360.85
  DATA FOR TRANSACTION  3

  PAY_DATE    = > 860331
  GROSS       = > 1360.50
  DATA FOR TRANSACTION  4

  EMP_ID      = >
```

You can place multiple repeating groups in the same statement. This PROMPT statement contains two repeating groups:

```
PROMPT EMP_ID 3 (PAY_DATE GROSS) 2 (DAT_INC SALARY)
```

The statement prompts for:

1. An employee ID.
2. A pay date and a monthly pay, three times.
3. A salary raise date (DAT_INC) and a new salary, two times.
4. The next employee ID.

You can nest repeating groups. For example, this prompt statement

```
PROMPT EMP_ID 6 (PAY_DATE 7 (DED_CODE DED_AMT))
```

prompts for:

1. An employee ID.
2. A pay date.
3. A deduction code and deduction amount, seven times.
4. Steps 2 and 3 repeat for a total of six times.
5. The next employee ID.

Syntax

How to Prompt Text

When you run a request containing PROMPT statements, the request prompts you for each field by displaying the field name and an equal sign (=). However, you may specify your own prompt. The syntax is

```
PROMPT fieldname.text.
```

where:

fieldname

Is the name of the field you are prompting for.

text

Is the text you want to appear as the prompt, up to 38 characters. Text must be enclosed within periods.

Note the following rules regarding prompt text:

- The text must be delimited by a period (.) on either side, with no space between the field name and the first period.
- The text cannot contain apostrophes or single quotation marks (').
- The text must be typed on one line.
- A single MODIFY request can contain up to 4000 characters of prompt text.

This request adds new employees to the EMPLOYEE data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID. ENTER THE EMPLOYEE ID NUMBER: .
PROMPT FIRST_NAME. ENTER FIRST NAME: .
PROMPT LAST_NAME. ENTER LAST NAME: .
PROMPT HIRE_DATE. WHAT DATE WAS EMPLOYEE HIRED? .
PROMPT CURR_SAL. WHAT IS THE STARTING SALARY? .

MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Special Responses

This section discusses special responses to prompts. It covers:

- Canceling a transaction.
- Ending execution.
- Correcting a field value.
- Typing ahead.
- Repeating the last response.
- Entering no data.
- Breaking out of repeating groups.
- Invoking the FIDEL Facility

Canceling a Transaction

To cancel a transaction, enter a dollar sign (\$) after any prompt. The request displays the following message

```
(FOC309) TRANSACTION INCOMPLETE:
```

and will prompt you for the next transaction. Canceling a transaction clears the buffer of data and causes the PROMPT statement to re-prompt you for the fields, allowing you to clear a bad transaction and start over.

Ending Execution

To end execution of the request, enter either Quit or End after any prompt. The request displays the execution statistics and returns you to the FOCUS command level. The data source will be updated to the last completed transaction.

Correcting Field Values

If you entered an incorrect field value, you can correct it at the next prompt. Type the value for the next prompt, but do not press Enter. Instead, type a comma and then type

```
fieldname = corrected-value
```

where *fieldname* is the field name of the corrected value. Then press Enter. Note that *fieldname* must be separated from the previous value by a comma.

The example below shows a user correcting a DEPARTMENT value after the CURR_JOBCODE prompt.

```
> DATA FOR TRANSACTION 1

EMP_ID          = > 071382660
DEPARTMENT      = > production
CURR_SAL        = > 19350.67
CURR_JOBCODE    = > a03, department=sales
DATA FOR TRANSACTION 2

EMP_ID          = >
```

Note: If you enter an incorrect field value at the last prompt of a transaction, you cannot correct the value in that transaction.

Typing Ahead

You can enter several values at one prompt by typing ahead. Enter

```
value-1, value-2, ... value-n
```

where:

```
value-1
```

Is the value of the field for which you are being prompted.

```
value-2 ...
```

Are the values of fields you have not yet been prompted for by the PROMPT statement. The values must be in the order of fields specified by the PROMPT statement, from the field being prompted for onwards. Separate the values with commas.

For example, a MODIFY request has this PROMPT statement:

```
PROMPT EMP_ID DEPARTMENT CURR_SAL CURR_JOBCODE
```

When you run the request, you enter an employee ID, a department, salary, and job code at the EMP_ID prompt, as shown below.

```
> DATA FOR TRANSACTION 1

EMP_ID          = > 071382660, sales, 23800, b04
DATA FOR TRANSACTION 2

EMP_ID          = >
```

Repeating a Previous Response

If you are going to respond to a prompt with the same value as the previous prompt, you may enter a double quotation mark (") instead to save typing.

Entering No Data

If you run a request that prompts you for a field that should not contain data, enter a period (.) after the prompt. The field becomes inactive and does not change any values in the data source.

If you are adding segments to the data source, the field in the new instance becomes:

- Blank, if the instance field is alphanumeric.
- Zero, if the instance field is numeric.
- The MISSING symbol, if the field is described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

Breaking Out of Repeating Groups

To break out of a repeating group, enter an exclamation point (!) after any prompt. The request will immediately prompt you for the first field outside the repeating group.

For example, you run this request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID 3 (PAY_DATE GROSS)

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE

MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Every time you enter an employee ID, the request prompts you for a pay date and a monthly pay (GROSS) three times. If you enter an exclamation point at one of these prompts, the request prompts you for the next employee ID.

Each cycle of prompts within a repeating group counts as one transaction. The repeating group data you entered before the transaction where you broke out remains active and modifies the data source.

If you break out of one repeating group nested in another repeating group, the request next prompts you for the fields of the outer group. For example, a request contains this PROMPT statement:

```
PROMPT EMP_ID 6 (PAY_DATE 7 (DED_CODE DED_AMT))
```

You run the request. If you enter an exclamation point at a DED_CODE or DED_AMT prompt, the request next prompts you for the next PAY_DATE value.

Reference PROMPT Phrases in MATCH and NEXT Statements

You can use PROMPT statements as phrases in MATCH or NEXT statements. By doing so, you avoid prompting the user for data that will be rejected anyway. The following examples illustrate the differences.

Consider the following request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL

MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

This request prompts the user for the EMP_ID and CURR_SAL fields. The MATCH statement searches the data source for the EMP_ID value the user enters (MATCH EMP_ID). If it finds the value, it updates the CURR_SAL value; otherwise it rejects the transaction. The user must enter both an EMP_ID and a CURR_SAL value every transaction, whether the transaction is accepted or not.

However, when the request prompts for the CURR_SAL value in the MATCH statement, the user enters a CURR_SAL value only if the corresponding EMP_ID value is in the data source. This request shows how this is done:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID

MATCH EMP_ID
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

The request prompts you for an EMP_ID value. It then searches the data source for the ID you entered. If it does not find the value, it rejects the ID and prompts you for another ID. Only if it finds the ID in the data source does it prompt you for a CURR_SAL value.

Reference Using PROMPT and FREEFORM Statements in One Request

You may use PROMPT and FREEFORM statements together in one request. This feature is useful when key field values are difficult to read and type, such as large numbers or complex codes. For example, a request might read employee ID numbers from a comma-delimited data source, use those IDs to locate segment instances, and then prompt the user for the data to update the employee information.

To use FREEFORM and PROMPT together, follow these rules:

- Place all FREEFORM statements before the PROMPT statements.
- Place the data in a separate data source. Specify the data source with the ON ddname option.
- Do not end the comma-delimited records with dollar signs (\$).

Note that when you use FREEFORM together with PROMPT, the transaction counter does not appear before the prompts.

This request updates employee salaries:

```
MODIFY FILE EMPLOYEE
FREEFORM ON EMPNO EMP_ID

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE "ENTER SALARY FOR EMPLOYEE #<EMP_ID"
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
DATA
```

Note the TYPE phrase in the MATCH statement that informs the user what employee ID the request is processing. The TYPE statement is described in *Displaying Specific Messages: The TYPE Statement* on page 9-117.

Invoking the FIDEL Facility: The CRTFORM Statement

This section is a brief description of the CRTFORM statement, which is discussed fully in Chapter 10, *Designing Screens With FIDEL*.

The CRTFORM statement invokes the FIDEL facility, which generates a formatted screen. You type the transaction values in the designated areas of the screen and press Enter.

To use the FIDEL facility, you must be on a full-screen terminal running FOCUS in interactive mode, not batch. Note that FIDEL is separate from the MODIFY facility, so your installation may have MODIFY but not FIDEL. Consult your systems manager or database administrator.

Beneath the CRTFORM statement, you specify the layout of the screen. Enclose each line of the screen in double quotation marks. On each line, you can type free text instructing the user and designate data entry areas where the user enters data for specific fields.

You may also display messages to the user in the TYPE area of the CRTFORM using the HELPMESSAGE attribute (see *Displaying Messages: Setting PF Keys to HELP* on page 9-132 and in the *Describing Data* manual).

The following request updates employees' department assignments, salaries, job codes, and classroom hours:

```

MODIFY FILE EMPLOYEE
CRTFORM
" ***** EMPLOYEE INFORMATION UPDATE *****"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"
"ENTER EMPLOYEE'S DEPARTMENT: <DEPARTMENT"
"ENTER CURRENT SALARY: <CURR_SAL"
"ENTER JOB CODE: <CURR_JOBCODE"
"ENTER CLASS HOURS: <ED_HRS"

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON MATCH UPDATE CURR_JOBCODE ED_HRS
DATA VIA FI3270
END

```

A request may have up to 255 CRTFORM statements, and may also have one FIXFORM statement preceding the CRTFORM statements. You may place CRTFORM phrases in MATCH and NEXT statements.

The FIDEL facility has several features that enhance its usability:

- Turnaround fields display field values as they exist in the data source, which you can then change.
- Display fields display field values that you cannot change. You can use these fields to design CRTFORM screens for data source inquiry.
- Screen attributes display different parts of the screen in different colors, highlighted, underlined, or flashing.
- Multiple-record processing allows you to modify several segment instances on one screen.

Please refer to Chapter 10, *Designing Screens With FIDEL*, to learn how to use FIDEL.

Entering Text Data Using TED

While in MODIFY, TED can be used to enter text field data. When TED is used to enter text, a new temporary file is opened in memory for data input; this file is never written to disk permanently. The name of this file is the same as the name of the text field. The file type in CMS and the ddname in TSO for the text field will be TXTFLD. For example

```
DESCRPT TXTFLD
```

is the file name and file type of the file opened for the text field DESCRPT.

All TED rules and functions apply, including the ability to edit other files. The RUN function in TED is ignored for text fields and is treated as the FILE command instead.

There are six ways to use the syntax for entering text format data using TED:

```
TED textfield
ON MATCH TED textfield
ON NOMATCH TED textfield
ON MATCH/NOMATCH TED textfield
ON NEXT TED textfield
ON NONEXT TED textfield
```

For example:

```
MODIFY FILE COURSES
PROMPT COURSE_CODE
MATCH COURSE_CODE
    ON NOMATCH TED DESCRIPTION
    ON NOMATCH INCLUDE
    ON MATCH TED DESCRIPTION
    ON MATCH UPDATE DESCRIPTION
DATA
```

TED will always edit the most recent version of the text field. The first time, this will be the current data source text field value; the next time that TED is used on the same text field, data from the previous text transaction will be available for editing.

As a rule, TED will always look for text data in the transaction area first. If no text exists there, TED looks for text present as a result of MATCH. If there is no data there, TED assumes that the field is new and brings up a new (empty) file.

After one transaction involving TED is complete, data areas are blanked out before proceeding with the next transactions (as when DEACTIVATE is used). This means that all text instances will be newly created (therefore, one course description will not carry over and accidentally be used for the next course number).

Text fields must always end with the end-of-text mark (%\$). Although you may enter this mark directly in the TED file as the first two characters on the last line, TED will test for the presence of the end-of-text mark; if it is missing, TED automatically inserts it.

Note: You must supply the end-of-text mark when using PROMPT or FIXFORM.

If you wish to use TED to input data for more than one text field, specify a separate action for each field:

```
ON MATCH TED TXFIELD1
ON MATCH TED TXFIELD2
```

The size of the file is limited only by the amount of available storage space.

Entering Text Field Data

The following rules apply to text field data entry using TED, FIXFORM, FREEFORM, or PROMPT:

- You can begin entering text data at any position on a line.
- Leading blanks on a line are preserved.

A line will be treated as the start of a new paragraph if it starts with three or more blanks. To prevent the concatenation of lines when a text field is displayed, insert at least three blanks at the beginning of each line.

- Blank lines are permitted.

Preserving Compatibility of Text Fields Using TED

You can use the SET TEXTFIELD command to preserve downward compatibility of text fields with prior FOCUS releases. The syntax is

```
SET TEXTFIELD = {OLD|NEW}
```

where:

OLD

Allows you to use text field data in prior releases of FOCUS when that data has been created or modified in Release 7.0. OLD is the default.

NEW

Disables the ability to use text field data in prior FOCUS releases when that data has been created or modified in Release 7.0.

In addition, FOCUS can preserve text fields exactly as entered into the data source using ON MATCH/NOMATCH TED.

Defining a Text Field

The syntax for defining a text field in a Master File is:

```
FIELD=fieldname, ALIAS=aliasname, FORMAT=TXnn, $
```

or

```
FIELD=fieldname, ALIAS=aliasname, FORMAT=TXnnF, $
```

where:

fieldname

Is the name you assign the text field.

aliasname

Is an alternate name for the field name.

nn

Is the output display length in TABLE for the text field.

F

Is used to format the text field for redisplay when TED is called using ON MATCH or ON NOMATCH. When F is specified, the text field is formatted as TX80 and is displayed. When F is not specified, the field is redisplayed exactly as entered.

Displaying Text Fields

FOCUS includes a format option in the text field of the Master File. Use of this determines whether text will display in the format in which it was entered.

For example, below is a Master File and the sample data that was entered into the field TXTFLD using TED.

```
FILE=TEXT, SUFFIX=FOC
  SEGNAME=SEGA, SEGTYPE=S1
  FIELD=KEYFLD, , A1, $
  FIELD=TXTFLD, , TX20, $
```

Sample data entered:

```
THIS IS A TEST OF THE NEW TED OPTION 'F'.  REMEMBER THAT TED DISPLAYS 80
CHARACTERS ON THE SCREEN.  THREE LEADING BLANKS ARE USED TO INDICATE A
NEW PARAGRAPH.  TEXT FIELD DATA IS ALWAYS STORED EXACTLY AS ENTERED.  WHEN
F IS INCLUDED IN THE FORMAT AND THE TEXT FIELD IS REDISPLAYED, BLANKS ARE
OMITTED AND THE FIELD IS CONDENSED.
WHEN F IS NOT INCLUDED, THE FIELD IS REDISPLAYED AS ENTERED.
```

Since the text field in the Master File does not include the F option, the data will be redisplayed exactly as entered using TED (ON MATCH TED TXTFLD).

For the next example, the text field includes the F option:

```
FILE=TEXT, SUFFIX=FOC
  SEGNAME=SEGA, SEGTYPE=S1
  FIELD=KEYFLD, , A1, $
  FIELD=TXTFLD, , TX20F, $
```

Note: The same data is entered as in the previous example.

In this case, since the text field does include the F option, when the field is redisplayed, blanks are omitted and the field is condensed as shown below:

```
THIS IS A TEST OF THE NEW TED OPTION 'F'.  REMEMBER THAT TED DISPLAYS 80
CHARACTERS ON THE SCREEN.  THREE LEADING BLANKS ARE USED TO INDICATE A
NEW PARAGRAPH.  TEXT FIELD DATA IS ALWAYS STORED EXACTLY AS ENTERED.
WHEN F IS INCLUDED IN THE FORMAT AND THE TEXT FIELD IS REDISPLAYED,
BLANKS ARE OMITTED AND THE FIELD IS CONDENSED.  WHEN F IS NOT INCLUDED,
THE FIELD IS REDISPLAYED AS ENTERED.
```

Specifying the Source of Data: The DATA Statement

The DATA statement marks the end of the executable statements in a request. It also specifies the source of the data.

Syntax **How to Use a DATA Statement**

`DATA [ON ddname|VIA program]`

where:

`ON ddname`

Indicates that the data is in a data source allocated to *ddname*.

`VIA program`

Indicates that the data is supplied directly from another computer program.

Type the DATA statement without parameters if:

- The data comes from the request itself.
- The request contains only PROMPT statements to read data.
- The request does not read any data (this occurs when you use a request to browse through a data source using the NEXT statement).

Reading Selected Portions of Transaction Data Sources: The START and STOP Statements

MODIFY requests read and process transaction data sources from the first record to the last. The START statement signals requests to read starting from a particular record in the data source. The STOP statement signals requests to stop reading at a particular record in the data source. You may use START and STOP statements to process transaction data sources in sections, to resume processing a transaction data source after a system crash, and to test a new request on a limited number of transactions.

Syntax How to Use a START Statement

```
START n
```

where:

```
n
```

Is the number of the first physical record to be processed by the request.

The syntax for the STOP statement is

```
STOP n
```

where:

```
n
```

Is the number of the last physical record to be processed by the request.

The START and STOP statements may appear anywhere in the request.

For example, the following request reads 300 records from a transaction data source (ddname SALDATE) starting from the 201st record until the 500th.

```
MODIFY FILE EMPLOYEE
START 201
STOP 500

FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA ON FIXSAL
END
```

Note that the numbers are that of physical records, not logical records, and that a request reads four physical records as one logical record. Assume each input record consists of four physical records. For example, if you want the request to read the data source starting from after the first ten transactions, type the START statement as

```
START 41
```

because 10 transactions are made up of 40 physical records.

If you are processing a large transaction data source, you may divide the processing into steps using the START and STOP statements. At the completion of each step, make a backup copy of the data source. If a step is aborted for any reason, you can use the last backup to restore the data source.

These two requests are the same. The first processes transactions 1 to 100,000. The second processes transactions 100,001 to 200,000:

```
MODIFY FILE EMPLOYEE
START 1
STOP 100000
FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
DATA ON FIXSAL
END
```

```
MODIFY FILE EMPLOYEE
START 100001
STOP 200000
FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
DATA ON FIXSAL
END
```

Modifying Data: MATCH and NEXT

The MATCH and NEXT statements are the core of MODIFY requests; they are the statements that determine which data source records are added, changed, or deleted. They work by selecting a particular segment instance, then updating or deleting it. They may also add new segment instances.

The MATCH statement selects specific segment instances based on their values. The NEXT statement selects the next segment instance after the current position.

The MATCH Statement

The MATCH statement selects specific segment instances based on their values. It compares one or more field values in the instances with corresponding incoming data values. The action it performs depends on whether there is a segment instance with matching field values.

For example, suppose a MODIFY request was processing this incoming data record in comma-delimited format

```
EMP_ID = 123456789, CURR_SAL = 20000, $
```

and that the request contained this MATCH statement:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH INCLUDE
```

This MATCH statement compares the EMP_ID value of an incoming data record to the EMP_ID values in segment instances:

- If a segment instance has EMP_ID value 123456789, the MATCH statement replaces the CURR_SAL value in the instance with the incoming CURR_SAL value of 20000.
- If there is no instance with the EMP_ID value of 123456789, the MATCH statement creates a new segment instance with the EMP_ID value of 123456789 and a CURR_SAL value of 20000.

Notice that the MATCH statement used each of the two incoming data fields differently. It used the EMP_ID field (specified after the word MATCH) to locate the segment instance (or to prove that it did not exist); it never altered the EMP_ID value in the segment. If it did locate the instance, it replaced the CURR_SAL value in the instance with the value in the incoming data field.

To identify the correct segment instance, the field values that the MATCH statement is searching for must be unique to the instance within its segment chain. For the most common types of segments, types S1 and SH1, the key field value is unique to each instance within its segment chain. This is the value you will usually be searching for.

Note that the MODIFY command cannot update key fields. To update key fields, use the FSCAN facility as described in Chapter 13, *Directly Editing FOCUS Databases With FSCAN*.

Remember from the introduction that FOCUS executes a MODIFY request for every transaction.

Syntax **How to Use a MATCH Statement**

```
MATCH { * [KEYS] [SEG n] | field1 [field2 field3 ... field-n] }  
  ON MATCH action-1  
  ON NOMATCH action-2  
  [ON MATCH/NOMATCH action-3]
```

where:

field1 ...

Are the names of incoming data fields to be compared with similarly named data source fields. The names may be full field names, aliases, or truncations. If a field value is missing, the value is treated as zeros for numeric fields and blanks for alphanumeric fields.

These fields are segment key fields unless the MATCH statement is modifying a segment of type S0 or blank. If the segment is type Sn or SHn and you do not specify the segment keys, the request adds the keys to the list automatically and displays a warning message.

If the list of fields is too long to fit on one line, begin each line with the word MATCH. For example:

```
MATCH EMP_ID DAT_INC TYPE  
MATCH PAY_DATE DED_CODE
```

To compare the values of all fields in the data source with incoming values, enter:

```
MATCH *
```

To compare the values of all key fields in the data source with incoming values, enter:

```
MATCH * KEYS
```

To compare the values of all key fields in a particular segment, type

```
MATCH * KEYS SEG n
```

where *n* is either the segment name or number as determined by the ? FDT query (described in the *Developing Applications* manual).

action-1

If the MATCH statement locates a segment instance with a data value matching the incoming data value (ON MATCH), it performs this action.

action-2

If the MATCH statement cannot locate a segment instance with a value matching the incoming data value (ON NOMATCH), it performs this action.

action-3

Whether or not the MATCH statement locates a segment instance with a value matching the incoming data value (ON MATCH/NOMATCH), it performs this action.

Note that you may include many ON MATCH and ON NOMATCH phrases in one MATCH statement. MATCH phrases can precede or follow NOMATCH phrases. The actions you may use in MATCH statements are listed in the section below. They fall into seven groups:

- Actions that modify segments.
- Actions that control MATCH processing.
- Actions that read incoming data fields.
- Actions that perform computations and validations or type messages to the terminal.
- Actions that control Case Logic.
- Actions that control multiple-record processing.
- Actions that activate and deactivate fields.

Please note the following rules regarding the MATCH statement:

- Each phrase of the MATCH statement must start on a separate line.
- The ON MATCH and ON NOMATCH phrases may be reversed.
- If an action has a list of fields, but the list of fields is too long to fit on one line, you may break the list into two or more lines. Begin each line with the ON MATCH or ON NOMATCH phrase, followed by the action. For example:

```
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON MATCH UPDATE CURR_JOBCODE ED_HRS
```

Syntax

How to Specify Actions With the ON MATCH/NOMATCH Phrase

The MATCH statement has an ON MATCH/NOMATCH phrase. This phrase specifies an action to be taken regardless of whether the field value for which the MATCH statement is searching exists in the data source. This phrase is especially useful when you are using CRTFORMs with display or turnaround fields (see Chapter 10, *Designing Screens With FIDEL*). For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
"ENTER EMPLOYEE'S ID: <EMP_ID"
MATCH EMP_ID
  ON MATCH/NOMATCH CRTFORM LINE 3
"ENTER DEPARTMENT: <T.DEPARTMENT"
"ENTER NEW SALARY: <T.CURR_SAL"
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON NOMATCH INCLUDE
DATA VIA FI3270
END
```

This request prompts you for an employee's ID. It then searches for the ID in the data source. It prompts you for the employee's new department and salary, whether the ID is in the data source or not. If the ID is in the data source, it updates the employee's department and salary; otherwise, it adds a new segment instance with the information.

You could not have placed the CRTFORM statement before the MATCH statement, because the CRTFORM statement contains turnaround fields.

You can specify the following actions in an ON MATCH/NOMATCH phrase:

- PROMPT
- TED
- CRTFORM
- GOTO
- IF
- ACTIVATE
- DEACTIVATE
- REPEAT
- HOLD

Note: TED in MODIFY can be used only with fields that have a text (TX) format (see *Entering Text Data Using TED* on page 9-52 for entering and editing text fields with TED).

Reference **MATCH Statement Defaults**

The following are defaults affecting the MATCH statement:

- If a MODIFY request has neither MATCH nor NEXT statements, it defaults to:

```
MATCH *
ON NOMATCH INCLUDE
```

It adds the instance even if another instance has the same key values. Since key values uniquely identify segments, you should avoid doing this unless you are loading data into a newly created data source, the incoming data is in a data source, and you know that there are no duplicate key values in the data.

The following request reads in data from a fixed-format data source, ddname EMPDATA, to load in data into the segments EMPINFO and SALINFO in the EMPLOYEE data source:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9   LAST_NAME/15  FIRST_NAME/10
FIXFORM PAY_DATE/I6   GROSS/D12.2
DATA ON EMPDATA
END
```

- If a MATCH statement has neither an ON MATCH nor an ON NOMATCH phrase, the MATCH statement defaults to:
- If a MATCH statement has an ON NOMATCH phrase but no ON MATCH phrase, the ON MATCH phrase defaults to:
- If a MATCH statement has a MATCH phrase but no NOMATCH phrase, the ON NOMATCH phrase defaults to:

```
ON MATCH CONTINUE
ON NOMATCH INCLUDE
```

```
ON MATCH CONTINUE
```

```
ON NOMATCH REJECT
```

Note: If a MATCH statement has the phrase

```
ON NOMATCH TYPE
```

and no other ON NOMATCH phrases, the request automatically adds the phrase:

```
ON NOMATCH REJECT
```

Adding, Updating, and Deleting Segment Instances

The most important function of the MATCH statement is the adding, updating, and deleting of segment instances. The MATCH statement does this by first searching a particular segment chain within a segment for specific instances (segment chains are groups of segment instances associated with an instance in the parent segment). The root segment contains just one segment chain; descendant segments are composed of many segment chains. How the MATCH statement selects segment chains in descendant segments is explained in *Modifying Data: MATCH and NEXT* on page 9-58.

The process can be summarized as follows:

1. The MODIFY request reads a transaction. The transaction contains values that identify a particular segment instance. Usually, these are key field values.
2. The MATCH statement searches the segment for an instance containing the key field values:

If it is adding a new instance, it must confirm that the instance is not yet in the segment. Otherwise, it would be adding a duplicate instance.

If it is updating or deleting an instance, it must first find the instance in the segment.

3. The MATCH statement takes action depending on whether it found the instance or not. These actions are as follows:

<code>ON NOMATCH INCLUDE</code>	The instance is not yet in the segment. Therefore, the request creates a new instance using values in the transaction.
<code>ON MATCH REJECT</code>	The new instance already exists in the segment. Therefore, the request does not add the instance to the data source. Rather, it rejects the transaction.
<code>ON MATCH UPDATE <i>list</i></code>	The instance exists in the segment. Therefore, the request changes the values of the data source fields named in <i>list</i> to the values in the transaction.
<code>ON MATCH DELETE</code>	The instance exists in the segment. Therefore, the request deletes the instance, all its descendants, and any references to the deleted instances in the indexes.
<code>ON NOMATCH REJECT</code>	The instance cannot be found in the segment. Therefore, it cannot be changed or deleted. The request rejects the transaction.

Example Adding Segment Instances

The syntax of a MATCH statement that adds segment instances is:

```
MATCH keyfield
  ON MATCH REJECT
  ON NOMATCH INCLUDE
```

When you include a new instance, the request fills the instance with the transaction field values. If some segment fields are absent in the transaction, they become blank or zeros in the instance, or the MISSING symbol if the field is described with the MISSING=ON attribute (discussed in the *Describing Data* manual).

FOCUS determines the placing of the instance within a segment chain based on the current position. The current position is the position of the instance you last added to the chain.

When FOCUS adds the next instance to a keyed segment, it determines whether the instance goes before or after the current position based on the sort order of the segment. If the instance goes after the current position, FOCUS matches field values from the current position forward until it finds the proper place for the new instance. If the instance goes before the current position, FOCUS matches field values from the beginning of the chain forward until it finds the place for the new instance.

To increase efficiency, submit your transactions in the same sorted order as the segment (ascending order for S_n segments, descending order for SH_n segments). This causes FOCUS to move through the chain in one direction only.

If you do not submit the transactions in sorted order, you may get this message:

```
WARNING..TRANSACTIONS ARE NOT IN SAME SORT ORDER AS FOCUS FILE
PROCESSING EFFICIENCY MAY BE DEGRADED
```

This condition indicates that data will not be loaded in an optimal manner.

The following request adds new instances to the root segment of the EMPLOYEE data source. The fields EMP_ID (the key field), LAST_NAME, and FIRST_NAME in the new instances are filled with incoming data values; the other fields are left zero or blank:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee's ID, last name, and first name.
2. You enter ID 071382660, last name SMITH, and first name HENRY.
3. The request determines whether ID 071382660 is in the segment. It is there, so the request rejects the transaction, displaying a message telling you so.
4. The request prompts you again for an employee's ID, last name, and first name.
5. You enter ID 123456789, last name SMITH, and first name HENRY.
6. The request determines whether ID 123456789 is in the segment. It is not there, so the request adds a new segment instance, with 123456789 as the key value, SMITH in the LAST_NAME field, and HENRY in the FIRST_NAME field. All other fields in the instance are blanks and zeros.

Example Updating Segment Instances

The syntax of a MATCH statement to update segment instances is

```
MATCH keyfield
  ON MATCH UPDATE list
  ON NOMATCH REJECT
```

where *list* is a list of data source fields to be updated using the values in the transaction. If the list of fields is too large to fit on one line, begin each line with the ON MATCH UPDATE phrase. For example:

```
ON MATCH UPDATE EMP_ID LN FN
ON MATCH UPDATE HDT DPT CSAL
ON MATCH UPDATE CJC OJT
```

To update all fields in a matched segment (except the key fields), type:

```
ON MATCH UPDATE * [SEG n]
```

Note: You cannot update key fields. To change key fields, use the FSCAN facility as described in Chapter 13, *Directly Editing FOCUS Databases With FSCAN*.

The following request updates the salary (CURR_SAL field) for employees you specify:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee's ID and a new salary.
2. You enter ID 123123123 and a salary of \$20,000.
3. The request searches the segment for ID 123123123 but cannot find the value. It rejects the transaction.
4. The request prompts you again for an employee ID and new salary.
5. You enter ID 071382660 and a salary of \$20,000.
6. The request finds ID 071382660 in the segment and changes the employee's salary to \$20,000.

You can combine adding and updating operations in one MATCH statement:

```
MATCH keyfield
  ON MATCH UPDATE field-1 field-2 ... field-n
  ON NOMATCH INCLUDE
```

This statement searches for a segment instance with a key field value the same as the similarly named incoming field value. If it finds the instance, it updates the instance. If it cannot find the instance, it adds a new instance. For example:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH INCLUDE
```

Example Deleting Segment Instances

The syntax of the MATCH statement for deleting a segment instance is:

```
MATCH keyfield
  ON MATCH DELETE
  ON NOMATCH REJECT
```

Note that the UPDATE action only updates fields when the transaction fields have values present.

This request deletes records of employees who have left the company:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON MATCH DELETE
  ON NOMATCH REJECT
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee ID.
2. You enter ID 987654321.
3. The request cannot find ID 987654321 in the segment, so it rejects the transaction, displaying a message telling you so.
4. The request prompts you for another employee ID.
5. You enter ID 119329144.
6. The request finds ID 119329144 and so on and deletes all record of the employee from the data source. This includes the employee's instance in the root segment and all descendant instances (such as pay dates, addresses, and so on).

Performing Other Tasks Using MATCH

You may specify actions in MATCH statements that can stand alone as statements elsewhere in the MODIFY request. These actions are: read incoming data, perform computations and validations, type messages, control Case Logic and multiple record processing, and activate and deactivate fields.

Note that the MATCH statement can perform several actions if the ON MATCH or ON NOMATCH condition occurs. To specify this, assign each action a separate ON MATCH or ON NOMATCH phrase. For example:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE "EMPLOYEE ID NOT FOUND"
  ON NOMATCH REJECT
```

There are two ON NOMATCH phrases in this request: one specifies the TYPE action, the other the REJECT action. If you include a REJECT action, it must appear last; otherwise the request will terminate and generate an error message.

Reference Reading Data

The following actions read incoming data. They work just as FIXFORM, FREEFORM, PROMPT, and CRTFORM statements:

<code>FIXFORM list</code>	Where <i>list</i> is a list of fields and formats. Reads in data from a fixed-format data source.
<code>FREEFORM list</code>	Where <i>list</i> is a list of incoming data fields. Reads in data from a comma-delimited data source.
<code>PROMPT list</code>	Prompts the user for data in fields named in <i>list</i> one field at a time.
<code>CRTFORM</code>	Prompts the user for data using the full-screen FIDEL facility. FIDEL is described in Chapter 10, <i>Designing Screens With FIDEL</i> .
<code>TED</code>	Opens a temporary file for text field data entry using TED.

Reference Computations, Validations, and Messages

The following actions perform calculations and validations and type messages. These actions work the same as the COMPUTE, VALIDATE, and TYPE statements:

<code>COMPUTE</code>	Performs computations.
<code>VALIDATE</code>	Performs validations.
<code>TYPE [ON ddname]</code>	Types messages to the terminal. When the ON ddname option is used, the messages are sent to a file defined by <i>ddname</i> .

Reference Controlling Case Logic

The following actions control Case Logic. They are discussed in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 9-137:

<code>GOTO casename</code>	Branches to another case named by <i>casename</i> .
<code>PERFORM casename</code>	Branches to another case named by <i>casename</i> , then returns to the PERFORM.
<code>IF expression [THEN] GOTO case1 [ELSE GOTO case2];</code>	If the expression is true, the request branches to the case named by <i>case1</i> ; otherwise the request branches to case named by <i>case2</i> .

Reference Controlling Multiple Record Processing

These actions control multiple-record processing and are described in *The REPEAT Method* on page 9-159:

<code>REPEAT</code>	Begins a REPEAT statement that executes a group of MODIFY statements repeatedly.
<code>HOLD list</code>	Where <i>list</i> is a list of data fields. Stores field values in a buffer.

Reference Activating and Deactivating Fields

These actions activate and deactivate fields as described in *Active and Inactive Fields* on page 9-199:

<code>ACTIVATE list</code>	Activates fields named in <i>list</i> .
<code>DEACTIVATE list</code>	Deactivates fields named in <i>list</i> .

Place these statements within a MATCH statement if you want to run them only when the request can locate incoming values in the data source (or confirm that incoming values are not in the data source). This improves efficiency and makes the request logic more flexible.

Example Using MATCH Actions in a Request

For example, assume you are designing a request to update employee salaries. Those employees who have spent more than 100 hours in class (the ED_HRS field) are granted an extra 3% bonus.

The particular data source you are updating only contains the records of a small number of company employees, but the transaction data source contains records for every employee in the company. If you place the COMPUTE statement calculating the bonuses by itself, it will calculate the bonus for every record in the transaction data source, whether or not the record will be accepted into the data source. Instead, use the COMPUTE statement as an ON MATCH option in a MATCH statement. COMPUTE will then calculate the bonus only for employees in the data source. The request is:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
    CURR_SAL = IF D.ED_HRS GT 100 THEN CURR_SAL*1.03
              ELSE CURR_SAL;
  ON MATCH UPDATE CURR_SAL
DATA
    
```


Note the use of a D. prefixed field in the COMPUTE expression (D.ED_HRS). This field refers only to ED_HRS values in the data source. You may refer to data source fields when using statements in MATCH and NEXT statements or after them. The data source fields must either be in the segment instance you are modifying or in a parent instance along the segment path.

Modifying Segments in FOCUS Structures

This section discusses how the MATCH command modifies segments other than the root segment. The section covers:

- Modifying unique segments.
- Modifying descendant segments.
- Modifying sibling segments (multi-path data sources).
- Modifying segments with no keys.
- Modifying segments with multiple keys.
- Using alternate views.

Reference Modifying Unique Segments

Unique segments are segments that consist of only one instance for every parent instance. They are always descended from other segments, but may not have descendants themselves. Because unique segment instances are extensions of their parent instances, they have no key fields.

There are two methods of modifying unique segments:

- The CONTINUE TO method allows you to add, update, and delete unique segment instances.
- The WITH-UNIQUES method allows you to add and update unique segment instances, but not to delete them. However, the WITH-UNIQUES method is easier to use.

Syntax **How to Modify Segment Instances Using the CONTINUE TO Method**

The CONTINUE TO method first locates the parent instance, then proceeds to the unique instance. The syntax of the MATCH command to modify unique segment instances using the CONTINUE TO method is:

```
MATCH keyfield  
  ON NOMATCH action-1  
  ON MATCH CONTINUE TO u-field  
  ON MATCH action-2  
  ON NOMATCH action-3
```

where:

keyfield

Is the key field of the parent segment instance.

action-1

Is the action the request performs if the parent instance cannot be found.

u-field

Is the name of any field in the unique child segment.

action-2

Is the action the request performs if a unique child instance exists.

action-3

Is the action the request performs if a unique child instance does not exist.

The actions that the request can perform are the same as those described in *Adding, Updating, and Deleting Segment Instances* on page 9-64 and *Performing Other Tasks Using MATCH* on page 9-68. The MATCH and NOMATCH phrases that follow the ON MATCH CONTINUE TO phrase can be in either order.

This example illustrates how the request selects unique segment instances. The root segment of the EMPLOYEE data source, called EMPINFO, which contains employee IDs, has a unique child segment called FUNDTRAN that contains information on employee bank accounts where pay checks are to be directly deposited. Every EMPINFO instance that describes an employee with a direct deposit bank account has one child instance in the FUNDTRAN segment.

You could prepare the following MODIFY request to enter information on employees that just opened a direct-deposit account:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts for an employee ID, bank name, and bank account number.
2. You enter employee ID 456456456, bank name BEST BANK, and bank account no. 235532.
3. The request does not find employee ID 456456456, so it rejects the transaction.
4. The request prompts you for another employee ID, bank name, and bank account number.
5. You enter employee ID 071382660, bank name BEST BANK, and bank account no. 235532.
6. The request finds ID 071382660. This employee has a segment recorded in the FUNDTRAN segment, meaning that the employee already has a direct-deposit bank account. The request rejects the transaction.
7. The request prompts you for another employee ID, bank name, and bank account number.
8. You enter employee ID 112847612, bank name BEST BANK, and bank account 235532.
9. The request finds employee ID 112847612 but finds no instance recorded for the employee in the FUNDTRAN segment.
10. The request records the bank name and bank account number in a new instance in the unique segment.

The following request updates direct-deposit account information:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH UPDATE BANK_NAME BANK_ACCT
  ON NOMATCH REJECT
DATA
```

The following request deletes account information for employees who have closed their direct-deposit accounts:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH DELETE
  ON NOMATCH REJECT
DATA
```

To modify multiple unique children of one instance using the CONTINUE TO method, use Case Logic as explained in *Case Logic Applications* on page 9-148.

Syntax

How to Process Unique Instances Using the WITH-UNIQUES Method

The WITH-UNIQUES method processes unique instances as extensions of their parents; that is, it considers a parent instance and its unique child as one instance. This method first searches for the parent instance. If it finds the parent, it can update the parent instance and create or update the unique child at the same time. If it does not find the parent, it can create the parent instance and the unique child at the same time.

The syntax for the MATCH statement using the WITH-UNIQUES method is

```
MATCH WITH-UNIQUES keyfield
  ON MATCH action1
  ON NOMATCH action2
```

where:

keyfield

Is the key field in the parent segment.

action1

Is the action performed if the MATCH statement locates the parent instance.

action2

Is the action performed if the MATCH statement does not locate the parent instance.

The MATCH statement can specify these actions:

- The INCLUDE action, which creates a new parent instance and unique children instances for which there is incoming data.
- The UPDATE action, which updates a parent instance and its unique children. If a child instance does not exist, FOCUS creates one.
- The DELETE action, which deletes the parent instance and all children instances.
- Actions that perform the functions listed in *Performing Other Tasks Using MATCH* on page 9-68.

Note that the WITH-UNIQUES method can add and update unique instances, but it cannot delete them without deleting the parent instance. To delete unique instances, use the CONTINUE TO method described in *How to Modify Segment Instances Using the CONTINUE TO Method* on page 9-72.

This MODIFY request adds information on new employees, including information on direct-deposit bank accounts. If an employee is already recorded in the data source, the request rejects the entire transaction. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID FIRST_NAME LAST_NAME
PROMPT BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
      ON MATCH REJECT
      ON NOMATCH INCLUDE
DATA
```

This MODIFY request updates employees' account information. If an employee just opened a direct-deposit account, the request automatically creates a new unique instance to record the information. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
      ON NOMATCH REJECT
      ON MATCH UPDATE BANK_NAME BANK_ACCT
DATA
```

This request adds and updates employees' account information, whether or not the employees are new:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME
PROMPT BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
    ON NOMATCH INCLUDE
    ON MATCH UPDATE BANK_NAME BANK_ACCT
DATA
```

Note that the WITH-UNIQUES method allows you to include and update the multiple unique children of one instance in one MATCH statement.

When using MATCH WITH-UNIQUES followed by ON MATCH COMPUTE, each computed field must have its own ON MATCH COMPUTE statement.

Example **Modifying Descendant Segments**

Modifying descendant segments is similar to modifying the root segment, with one difference: when a MATCH statement searches a root segment for a key field value, it searches every instance of the segment. When the MATCH statement searches a descendant segment, however, it searches only the segment chain belonging to a particular parent instance. If the MATCH statement cannot find the key field value in this chain, it executes the ON NOMATCH phrase. To modify the chain, you must first identify the parent instance using a previous MATCH statement.

The following example illustrates this. The EMPLOYEE data source contains two segments: An EMPINFO segment containing employee IDs, and a child segment called SALINFO that keeps track of each employee's monthly pay. Each of these IDs has an instance in the SALINFO segment for each month that the employee worked (for example, an employee working for eight months has eight instances in the SALINFO segment).

To modify a June instance in the SALINFO segment, you must first identify which employee was paid in June. If the MODIFY request cannot find the June instance for one employee, it will execute the ON NOMATCH phrase even though a June instance exists for another employee.

This request adds a new monthly pay instance for each employee in the company. Note the word CONTINUE, which causes the request to proceed to the next MATCH statement (which adds the instances to the descendant segment) without taking any action. Also note that the phrase ON NOMATCH CONTINUE is illegal:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH CONTINUE
MATCH PAY_DATE
    ON MATCH REJECT
    ON NOMATCH INCLUDE
DATA
```

An execution might go as follows:

1. The request prompts you for an employee ID, the date the employee was paid, and the gross earnings paid.
2. You enter an employee ID 159159159, pay date 820831 (August 31, 1982), and gross earnings of \$916.67.
3. The request cannot find ID 159159159, so it rejects the transaction.
4. The request prompts you for another employee ID, pay date, and gross earnings.
5. You enter employee ID 071382660, pay date 820831, and gross earnings of \$916.67.
6. The request finds ID 071382660, and searches the SALINFO segment chain belonging to 071382660 for the pay date 820831.
7. The request finds the pay date 820831 in the segment chain. Since the instance already exists, the request rejects the transaction.
8. You enter employee ID 071382660, pay date 820930 (September 30, 1982), and gross earnings of \$916.67.
9. The request finds ID 071382660, and searches the SALINFO segment chain belonging to 071382660 for the pay date 820930.
10. The request does not find pay date 820930 in the segment chain, so it includes a new instance in the SALINFO segment chain for pay date 820930 with gross earnings of \$916.67.

Modifying Data: MATCH and NEXT

If your request prompts for data (using either PROMPT or CRTFORM), it is better to prompt for the child key field values after the request locates the parent key field values. This spares the user from typing the child key if the request cannot locate the parent key. You can rewrite the previous request as:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT PAY_DATE GROSS
MATCH PAY_DATE
    ON MATCH REJECT
    ON NOMATCH INCLUDE
DATA
```

You can also write the request to include a new EMPINFO segment instance and a new SALINFO instance if the employee's ID is not already there:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
    ON NOMATCH INCLUDE
    ON MATCH CONTINUE
MATCH PAY_DATE
    ON NOMATCH INCLUDE
    ON MATCH REJECT
DATA
```

The first MATCH statement searches the EMPINFO statement for the employee ID that you entered. If it does not find the ID, the request creates a new EMPINFO segment instance with the new ID, and a descendant SALINFO instance with the pay date and monthly pay you entered.

Note that when an INCLUDE action creates a new segment instance, it also creates all descendant instances for which data is present.

If the employee ID is already in the data source, the second MATCH statement searches the SALINFO segment for the pay date you entered. If it does not find the ID, the request creates a new SALINFO instance with the pay date. If the pay date is already in the segment, the request rejects the transaction.

This request updates monthly pay instances:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT PAY_DATE GROSS
MATCH PAY_DATE
    ON MATCH UPDATE GROSS
    ON NOMATCH REJECT
DATA
    
```

This request deletes monthly pay instances:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT PAY_DATE
MATCH PAY_DATE
    ON MATCH DELETE
    ON NOMATCH REJECT
DATA
    
```

You may combine the MATCH statements in the request into one statement. This is called matching across segments. To match across segments, specify the key fields that the request must search for from the root segment down to the descendant segment (in that order) after the MATCH keyword. For example, the request above that updates employee's monthly pay can be rewritten this way:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID PAY_DATE
    ON NOMATCH REJECT
    ON MATCH UPDATE GROSS
DATA
    
```

This is the request shown earlier in this section that adds data on new employees and employees' monthly pay:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
    ON MATCH CONTINUE
    ON NOMATCH INCLUDE
MATCH PAY_DATE
    ON MATCH REJECT
    ON NOMATCH INCLUDE
DATA
    
```

This request can be rewritten this way:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Note: When a MATCH statement matches across segments, the explicit ON MATCH and ON NOMATCH phrases in the statement are only executed for the last descendant segment (key field PAY_DATE in the example). For the other segments, the request executes default phrases. If you are updating or deleting instances, these phrases are:

```
ON MATCH CONTINUE
ON NOMATCH REJECT
```

If, for example, you include an ON NOMATCH TYPE phrase in the MATCH statement, the phrase only types a message when there is an ON NOMATCH condition on the last segment.

If you are adding new instances, the default phrases are:

```
ON MATCH CONTINUE
ON NOMATCH INCLUDE
```

Because of these defaults, use this technique only when you are confident that you understand the logic of the request.

Example **Modifying FOCUS Structures of Three or More Levels**

What has been said for two-level FOCUS structures is true for three or more levels. To modify a descendant segment instance, you must first identify the parent instances to which the descendant instance belongs, from the root segment down to the immediate parent segment (the descendant segment instance belongs to a parent instance, that instance belongs to grandparent instance, and so on up the FOCUS structure to one of the root instances).

The following request illustrates this. The SALINFO segment has a child segment called DEDUCT that records all the different deductions that are taken from each monthly wage. If four deductions are taken from a monthly pay, that pay has four instances in the DEDUCT segment. The key field in the DEDUCT segment is DED_CODE; it specifies the type of deduction, such as certain taxes. The amount of the deduction is contained in the field DED_AMT.

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE DED_CODE DED_AMT
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH CONTINUE
MATCH PAY_DATE
    ON NOMATCH REJECT
    ON MATCH CONTINUE
MATCH DED_CODE
    ON NOMATCH REJECT
    ON MATCH UPDATE DED_AMT
DATA
    
```

Example **Modifying Sibling Segments (Multi-Path Data Sources)**

If you are modifying sibling segments (segments that have a common parent), place the MATCH statements modifying the siblings in any order after the MATCH statement identifying the parent instance. Each sibling must have a separate MATCH statement. If you are modifying descendants of one of the siblings, the MATCH statements that modify the children should follow immediately after the MATCH statement that identifies the sibling.

The following request updates the SALINFO and ADDRESS segments, both children of the EMPINFO segment. The ADDRESS segment contains both home and bank addresses of the employees; its key field is TYPE, which indicates whether the address is a home address or a bank address.

The request is as follows:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT PAY_DATE GROSS TYPE ADDRESS_LN1
MATCH PAY_DATE
    ON NOMATCH REJECT
    ON MATCH UPDATE GROSS
MATCH TYPE
    ON NOMATCH REJECT
    ON MATCH UPDATE ADDRESS_LN1
DATA
```

Syntax How to Modify Segments With No Keys

Segments of types S0 and blank (SEGTYPE=) have no key fields. Segments of type blank are always descendant segments; they can never be root segments. Segments of type S0 can be root segments.

To modify these segments, the MATCH statement selects instances by comparing the values of one or more fields in the segment to a similarly named transaction field. The MATCH statement has the form

```
MATCH {* [SEG n] | field-1 field-2 ... field-n}
      ON MATCH action-1
      ON NOMATCH action-2
```

where:

field-1 ...

Are any fields in the segment you are modifying.

* SEG *n*

Matches all fields in the segment, where *n* is either the segment name or number as determined by the ? FDT query (described in the *Developing Applications* manual).

The difference between segment type S0 and blank is in the way FOCUS adds new instances to the segments.

Example Storing Data With Type S0 Segments

When you add a segment instance to a type S0 segment, FOCUS matches field values in the segment chain from the current position forward through the chain, inserting the instance in the chain based on ascending order. FOCUS does not search the chain from the beginning; therefore, if the instance belongs before the current position, FOCUS inserts the instance at the end of the chain (this means that if you are adding instances to a new segment chain, FOCUS stores the instances in the order of submission). It may insert the instance even if another instance has the same field values and you specified ON MATCH REJECT. If, however, you sort the transactions in ascending sequence before submitting them, you will preserve the correct sequence in the chain. You will also prevent adding duplicate segments unless you specify ON MATCH INCLUDE.

Because it is difficult to ensure that segments of type S0 do not have instances with duplicate field values, they are difficult to maintain. You should only use them for data that needs to be loaded in once and does not need to be changed or deleted.

This is a sample FOCUS data source that stores memos, called MEMO. The Master File is:

```
FILE=MEMO , SUFFIX=FOC , $
SEGMENT=MEMOSEG , SEGTYPE=S1 , $
  FIELD=MEMO_NAME , ALIAS=MEMO , FORMAT=A25 , $
SEGMENT=TEXTSEG , SEGTYPE=S0 , PARENT=MEMOSEG , $
  FIELD=LINE , ALIAS=LN , FORMAT=A70 , $
```

The following request enters ten-line memos into the data source:

```
MODIFY FILE MEMO
PROMPT MEMO_NAME 10 (LINE)
MATCH MEMO_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
MATCH LINE
  ON MATCH INCLUDE
  ON NOMATCH INCLUDE
DATA
```

Note: The INCLUDE action in both ON MATCH and ON NOMATCH phrases adds a line of text even if the line is the same as another line in the memo (which would happen if you have more than one blank line in the memo) in all circumstances.

Reference **Type Blank Segments**

When you add an instance to a type blank segment, the MODIFY request compares the instance you are adding to every instance in the segment chain, based on the fields you specify in the MATCH statement. Thus, if you specified the ON MATCH REJECT phrase in the MATCH statement, the request does not allow you to add an instance that has the same field values you are matching on as another instance.

You modify type blank segments the same way you modify other segments. Be careful, however, that the fields you are matching on uniquely identify the segment instances, or you may not be able to select the instance you want to modify. (MODIFY requests always select the first instance that fulfills the match conditions.)

Example Modifying Segments With Multiple Keys

Segments may have multiple keys. These segments are types Sn or SHn where *n* is the number of keys. For example, a segment in ascending order that has two keys is type S2; that is, it has the attribute SEGTYPE=S2 in the Master File. Multiple keys are necessary when the first field alone cannot uniquely identify a segment instance. For example, a segment has three fields as described by the Master File:

```
FILE=ADDRESS ,SUFFIX=FOC , $
  SEGMENT=ADDRSEG ,SEGTYPE=S2 , $
    FIELD=LAST_NAME ,ALIAS=LNAME ,FORMAT=A15 , $
    FIELD=FIRST_NAME ,ALIAS=FNAME ,FORMAT=A15 , $
    FIELD=ADDRESS ,ALIAS=ADDR ,FORMAT=A80 , $
```

Since LAST_NAME field is not enough to identify individual segment instances (some people share the same last name), the segment uses the first two fields, LAST_NAME and FIRST_NAME, as keys.

Note that multiple keys must always be the first fields in the segment, and they must be next to each other; that is, a non-key field cannot be between two key fields.

Modifying segments with multiple key fields is the same as modifying segments with one key field. The one difference is that you must specify all the key fields in the MATCH phrase.

To enter data into the ADDRESS data source, you prepare the following MODIFY request:

```
MODIFY FILE ADDRESS
PROMPT LAST_NAME FIRST_NAME ADDRESS
MATCH LAST_NAME FIRST_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts you for the last name, first name, and address.
2. You enter last name FOX, first name GEORGE, and address 2365 N. HAMPTON ST. HAMILTON, MN 55473.
3. The request searches the segment for an instance with both last name FOX and first name GEORGE.
4. The request does not find such an instance, so it creates a new instance for George Fox.

Note that you cannot update any of the key fields.

Syntax **How to Use Alternate File Views**

To modify descendant segments, you must first specify the parent segments using a series of MATCH statements. You can modify a descendant segment directly by declaring the segment to be the root segment of an alternate file view. To do this, the segment must fulfill three conditions:

- The segment must be type S1 or SH1.
- The key field must be indexed.
- The key field values should be unique throughout the data source.

To declare an alternate file view, you begin the MODIFY request this way

```
MODIFY FILE filename.field
```

where:

filename

Is the name of the FOCUS data source you are modifying.

field

Is the name of the indexed key field in the root segment of the alternate file view.

Note that you can only update the root segment of the alternate file view; you cannot add or delete segment instances. However, you can add, update, and delete segment instances in the descendants of this segment. In addition, you may make use of external indices only using the FIND and LOOKUP functions. Be aware that an external index cannot be used as an entry point. For example,

```
MODIFY FILE filename.field
```

will be ineffective. FIND and LOOKUP are described in *Special Functions* on page 9-107.

This sample FOCUS data source, called BANK, contains information on bank accounts. The Master File is:

```
FILE=BANK , SUFFIX=FOC , $
SEGMENT=CUSTSEG , $
    FIELD=SOC SEC NUM , ALIAS=SSN , FORMAT=A9 , $
    FIELD=NAME , ALIAS=NAME , FORMAT=A30 , $
SEGMENT=ACCTSEG , SEGTYPE=S1 , PARENT=CUSTSEG , $
    FIELD=ACCT NUM , ALIAS=ACCOUNT , FORMAT=A10 , $
    FIELDTYPE=I , $
FIELD=AMOUNT , ALIAS=AMOUNT , FORMAT=D10.2 , $
SEGMENT=TRANSSEG , SEGTYPE=S1 , PARENT=ACCTSEG , $
    FIELD=TRANSNUM , ALIAS=TNUM , FORMAT=I5 , $
    FIELD=TRANTYPE , ALIAS=TTYPE , FORMAT=A1 , $
    FIELD=TR_AMOUNT , ALIAS=TAMOUNT , FORMAT=D8.2 , $
```


This Description contains three segments:

- The CUSTSEG segment contains social security numbers and names of bank depositors.
- The ACCTSEG segment, child of CUSTSEG, contains account numbers and the amount of money in each account. Note that the field ACCT_NUM is indexed and that each account number is unique throughout the data source.
- The TRANSSEG segment, child of ACCTSEG, contains information on individual bank account transactions: the transaction serial number (TRANSNUM), the type of transaction (TRANTYPE, which contains a D for deposits and a W for withdrawals), and the amount of the transaction (TR_AMOUNT).

To add new account information in the BANK data source, prepare the following MODIFY request:

```
MODIFY FILE BANK
PROMPT SSN NAME ACCT_NUM AMOUNT
MATCH SSN
    ON NOMATCH INCLUDE
    ON MATCH CONTINUE
MATCH ACCT_NUM
    ON NOMATCH INCLUDE
    ON MATCH REJECT
DATA
```

The MODIFY request above first specifies the parent segment CUSTSEG (MATCH SSN) before the child segment ACCTSEG (MATCH ACCT_NUM). Since ACCTSEG is an S1 segment with an indexed key field (ACCT_NUM), you can modify the ACCTSEG directly with this request:

```
MODIFY FILE BANK.ACCT_NUM
PROMPT ACCT_NUM AMOUNT
MATCH ACCT_NUM
    ON NOMATCH REJECT
    ON MATCH UPDATE AMOUNT
DATA
```

You may modify the root segment of the alternate file view and its descendants in the original data source structure, but not its parents. In the BANK data source, you may modify the TRANSSEG segment using the above alternate file view but not the CUSTSEG segment.

This request adds information on new bank account transactions to the data source:

```
MODIFY FILE BANK.ACCT_NUM  
PROMPT ACCT_NUM AMOUNT PROMPT TRANSNUM TRANTYPE TR_AMOUNT  
MATCH ACCT_NUM  
    ON NOMATCH REJECT  
    ON MATCH UPDATE AMOUNT  
MATCH TRANSNUM  
    ON MATCH REJECT  
    ON NOMATCH INCLUDE  
DATA
```

Selecting the Instance After the Current Position: The NEXT Statement

The NEXT statement selects the next segment instance after the current position, making the instance the new current position. The current position depends on the execution of MATCH and NEXT statements:

- If a MATCH or NEXT statement selects a segment instance, the instance becomes the current position within the segment.
- If a MATCH or NEXT statement selects a parent instance of a segment chain, the current position is before the first instance in the chain.
- At the beginning of a request, the current position in the root segment is before the first instance.

The NEXT statement can modify segment instances similarly to the MATCH statement and follows the same rules (see *The MATCH Statement* on page 9-59). However, the NEXT statement is most often used for displaying data source values.

Syntax **How to Use a NEXT Statement**

The syntax of the NEXT statement is

```
NEXT field
  ON NEXT action-1
  ON NONEXT action-2
```

where:

field

Is any field in the segment whose instances are being selected.

action-1

Is the action the request takes if there is a next instance to select.

action-2

Is the action the request takes if it has reached the end of the segment chain.

There can be many ON NEXT and ON NONEXT phrases in a single NEXT statement. Each phrase specifies one action.

An action can be any action that is legal in the MATCH statement (see *Adding, Updating, and Deleting Segment Instances* on page 9-64 and *Performing Other Tasks Using MATCH* on page 9-68). However, use ON NEXT INCLUDE and ON NONEXT INCLUDE phrases only to add instances to segments of type S0 or blank. If you use these phrases to modify other segments, you may duplicate what is already there. The difference between the two phrases is:

- ON NEXT INCLUDE adds a new segment instance after the current position.
- ON NONEXT INCLUDE adds a new instance at the end of the segment chain. The phrase ON NEXT INCLUDE is only valid for segments with type S0 or blank.

The following phrases are always illegal:

```
ON NONEXT UPDATE
  ON NONEXT DELETE
  ON NONEXT CONTINUE
  ON NONEXT CONTINUE TO
```

This phrase is legal even in requests that do not involve Case Logic:

```
ON NONEXT GOTO EXIT
```

The phrase terminates the request when the NEXT statement reaches the end of the segment chain.

Note that a NEXT statement can have multiple ON NEXT and ON NONEXT phrases. For example, the following statement displays the salaries of every employee in the data source and shows what their salaries would be if they are granted a 5% increase:

```
NEXT EMP_ID
  ON NEXT COMPUTE NEWSAL = 1.05 * D.CURR_SAL;
  ON NEXT TYPE
    "EMPLOYEE <D.EMP_ID SALARY NOW:<D.CURR_SAL"
    "SALARY PLUS 5% INCREASE: <NEWSAL"
  ON NONEXT TYPE
    "END OF EMPLOYEE FILE"
  ON NONEXT GOTO EXIT
```

Example **Selecting Instances**

You can use NEXT statements in non-Case Logic requests to modify or display the data in:

- The entire root segment.
- The first instances of segment chains in descendant segments.

To modify or display data in *entire* descendant segment chains, you must use Case Logic as described in *Case Logic Applications* on page 9-148.

The NEXT statement can modify and display data in the root segment. This request displays all the employee IDs in the employee ID segment:

```
MODIFY FILE EMPLOYEE
NEXT EMP_ID
  ON NEXT TYPE "EMPLOYEE ID: <D.EMP_ID"
  ON NONEXT GOTO EXIT
DATA
```

When a NEXT statement modifies or displays data in a descendant segment, it can do so only to the first instance in a segment chain. Consider the following request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE "YOU ENTERED ID <EMP_ID"

NEXT PAY_DATE
  ON NEXT TYPE
    "THIS EMPLOYEE'S LAST PAY DATE"
    "WAS <D.PAY_DATE"
  ON NONEXT GOTO EXIT
DATA
```

The MATCH statement selects an instance with a particular employee ID. The NEXT statement selects the instance with the employee's last pay date (the pay dates are organized in the data source from high to low). The PAY_DATE segment is a child of the EMP_ID segment.

The NEXT statement is at its most powerful when it is used to browse through an entire chain. To browse through a chain in a descendant segment, you must use Case Logic, as described in *Case Logic Applications* on page 9-148.

Displaying Unique Segments

You can use the NEXT statement to display and modify the contents of unique segments using two methods (see *Modifying Segments in FOCUS Structures* on page 9-71):

- The CONTINUE TO method.
- The WITH-UNIQUES method.

Syntax How to Use the CONTINUE TO Method

The syntax of the CONTINUE TO method is

```
NEXT field
  ON NONEXT action-1
  ON NEXT CONTINUE TO u-field
    ON NEXT action-2
    ON NONEXT action-3
```

where:

field

Is the first field in the parent instance.

action-1

Is the action the request performs if there are no more instances in the parent segment chain.

u-field

Is the name of any field in the unique child segment.

action-2

Is the action the request performs if the parent instance has a unique child instance.

action-3

Is the action the request performs if the parent instance does not have a unique child instance.

Syntax **How to Use the WITH-UNIQUES Method**

The syntax of the WITH-UNIQUES method is

```
NEXT WITH-UNIQUES field  
  ON NONEXT action1  
  ON NEXT action2
```

where:

field

Is the name of any field in the parent segment.

action1

Is the action the request performs if there are no more instances in the chain.

action2

Is the action the request performs if there is a next instance in the chain. This action can be performed on either the parent instance or the unique instance. If an UPDATE action updates a unique instance that does not exist yet, FOCUS creates the instance.

Computations: COMPUTE and VALIDATE

The MODIFY command provides two facilities that perform calculations on incoming data fields, data source fields, and temporary fields. These are:

- The COMPUTE statement. This statement allows you to modify incoming data field values and to define temporary fields.
- The VALIDATE statement. This statement allows you to reject transactions that contain unacceptable values.

FIND and LOOKUP functions can be used only in COMPUTE and VALIDATE statements. For more information, see *Special Functions* on page 9-107.

Computing Values: The COMPUTE Statement

The COMPUTE statement allows you to modify incoming data field values and to define temporary fields.

A transaction data source (whether stored on the computer or typed on paper) used to modify a data source often does not contain the same data that is to go into the data source fields. There are many reasons for this:

- The incoming data contains short codes representing the alphanumeric data that is to go into the data source. For example, incoming records contain the code P for PRODUCTION and M for MIS. The PRODUCTION and MIS values update the DEPARTMENT field.
- The incoming data is repetitive: the same value is used to update each instance or the same series of values is used to update each segment chain. For example, all employees are to receive a pay increase of 5%.
- The incoming data values are calculable from other values. For example, an employee's percentage salary increase is equal to the new salary divided by the old salary minus 1.
- Some values vary in predictable ways depending on other values. For example, employee salary increases depend on the employees' department assignment.

The COMPUTE statement gives you control over the data that modifies the data source. Using COMPUTE you can:

- Translate codes into data to modify the data source.
- Adjust the values of transaction fields.
- Define a data value or a series of data values to modify the data source repeatedly.
- Calculate data values from other sources and use these new values to modify the data source.

The COMPUTE statement works by setting either an incoming data field or a temporary field to the value of an expression. The expression may involve existing data source fields, other temporary fields, and constants.

Note that there are three different types of fields:

- Incoming data fields (also called transaction fields) contain data read from transaction data sources or a terminal. These fields are specified by the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements. They remain incoming data fields even if their values are changed by COMPUTE statements.
- Data source fields contain data stored in the data source. Their field names are prefaced by the D. prefix.
- Temporary fields are created by and receive their values from COMPUTE statements.

The following request uses all three types of fields. The request awards a bonus of \$150 to employees who received salary raises:

```
MODIFY FILE EMPLOYEE
1. PROMPT EMP_ID CURR_SAL
   COMPUTE
2.   BONUSAL/D8.2 = CURR_SAL + 150;
   MATCH EMP_ID
     ON NOMATCH REJECT
     ON MATCH COMPUTE
3.   CURR_SAL = IF CURR_SAL GT D.CURR_SAL
               THEN BONUSAL
               ELSE CURR_SAL;
   ON MATCH UPDATE CURR_SAL
DATA
```

The numbers above refer to these fields:

1. The EMP_ID and CURR_SAL fields are incoming data fields, because they are read by a PROMPT statement.
2. The BONUSAL field is a temporary field, because it is created by and receives its value from a COMPUTE statement.
3. The D.CURR_SAL field is a data source field, since its field name is prefaced with the D. prefix.

You may use COMPUTE statements to adjust the values of incoming data fields. For example, your MODIFY request reads salary values from a data source and places them into the field SALARY. You want to increase all these values by 10%. To do so, add this statement to the request:

```
COMPUTE SALARY = SALARY * 1.1;
```

In cases where the same field name exists in more than one segment, and that field must be redefined, the REDEFINES command should be used.

You may use the COMPUTE statement to define an unlimited number of temporary fields. For example, you define a temporary field TEMPSAL to contain the number 25000 if an employee is in the MIS department and the number 18000 if an employee is in the PRODUCTION department:

```
COMPUTE
  TEMPSAL =IF DEPARTMENT IS 'MIS' THEN 25000
           ELSE IF DEPARTMENT IS 'PRODUCTION' THEN 18000;
```


Note that MODIFY requests allow the use of up to 3,072 fields within the request. The number includes:

- Data source fields referred to in the request.
- Temporary fields created by COMPUTE and VALIDATE statements.
- Temporary fields created automatically by FOCUS. These include:
 - FOCURRENT for MODIFY requests run in Simultaneous Usage mode. FOCUS creates one FOCURRENT variable per request.
 - REPEATCOUNT for MODIFY requests containing REPEAT statements. FOCUS creates one REPEATCOUNT variable per request regardless of the number of REPEAT statements.
 - HOLDCOUNT and HOLDINDEX for MODIFY requests containing HOLD statements. FOCUS creates one HOLDCOUNT and one HOLDINDEX variable per request regardless of the number of HOLD statements.

Each field referred to or created in a MODIFY request counts as one field toward the 3,072 total, regardless of how often its value is changed by COMPUTE and VALIDATE statements. However, if a data source field is read by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement and also has its value changed by COMPUTE and VALIDATE statements, it counts as two fields.

FOCUS compiles most COMPUTE and DEFINE calculations when the request is parsed. Typically, the new compilation logic executes the compiled calculations in about one-fifth the time required by uncompiled calculations. However, the compiled form requires more memory. For this reason, very large MODIFY procedures may require more virtual storage to run and, should the MODIFY procedures be compiled, they will occupy more disk space.

There are two places in the MODIFY request where you can use COMPUTE statements:

- At the beginning of the request. COMPUTE statements here define temporary field values for every transaction. Note that these statements may not perform calculations on data source field values (D. fields).
- In or following MATCH and NEXT statements. COMPUTE statements here define temporary field values for transactions depending whether or not the MATCH or NEXT statement selected a particular segment instance. These statements may perform calculations using data source field values.

This section covers:

- The syntax of COMPUTE statements.
- Use of COMPUTE statements in MATCH and NEXT statements.
- Modifying transaction fields.
- Defining non-data source transaction fields.

Syntax **How to Use a COMPUTE Statement**

The syntax of the COMPUTE statement is as follows (note that you can place several COMPUTE statements after the COMPUTE keyword):

```
COMPUTE  
field[/format] = expression;  
field[/format] = expression;  
.  
.  
.
```

where:

field

Is the name of the field being set to the value of *expression*. The field can be an incoming data field or it can be a temporary field (whose name must be different from the incoming field names). Fields can only modify data source fields with the same name.

format

Is the format of the field if the field is temporary. Specify the format when defining the temporary field for the first time. Field formats are described in the *Describing Data* manual.

You can specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

You can specify the YRTHRESH and DEFCENT options to handle cross-century dates. Using these options, and working with cross-century dates, is discussed in the *Developing Applications* manual.

expression;

Is any expression valid in a DEFINE or TABLE COMPUTE statement. In addition, you may use the FIND and LOOKUP functions, described in *Special Functions* on page 9-107.

Note: The expression can be null; that is, the COMPUTE statement can have the form

```
COMPUTE  
field/format=;
```

where *format* is the format of the field. This form is used to define transaction fields that are not listed in the Master File.

Note that you must terminate the expression with a semi-colon (;). You may type a COMPUTE statement over as many lines as you need, terminating the expression with a semi-colon. The COMPUTE command supports other attributes such as DFC, YRT, and MISSING. See the *Creating Reports* manual for details.

For example:

```
COMPUTE
CURR_SAL = IF CURR_JOBCODE IS A02 THEN 15000
           ELSE IF CURR_JOBCODE IS B02 THEN 17000
           ELSE IF CURR_JOBCODE IS B12 THEN 18000
           ELSE 20000;
```

In the preceding example, the temporary field CURR_SAL will contain 15000, 17000, 18000, or 20000, depending on the value of CURR_JOBCODE. CURR_SAL will then be used later in the MODIFY request.

You can also place an expression on the same line as a COMPUTE keyword, and several expressions on one line (ending each expression with a semicolon). For example:

```
COMPUTE CURR_SAL=CURR_SAL*1.2; ED_HRS = ED_HRS-5;
```

You can specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

Example Placing COMPUTE Phrases in MATCH and NEXT Statements

You may place COMPUTE statements in MATCH and NEXT statements. The request only performs the computation if the MATCH or NEXT condition is met. These COMPUTE phrases may perform calculations on data source field values if these fields are either in the segment instance being modified or in a parent instance along the segment path (the parent instance, the parent's parent, and so on until the root segment). To specify data source field values (as opposed to values in the transaction field with the same name), affix the D. prefix to the front of the field name.

Note that COMPUTE statements that follow a MATCH or NEXT statement may also perform calculations on data source field values if these fields are in the instance selected by the previous statement (or are in the segment path).

When using MATCH WITH-UNIQUES followed by ON MATCH COMPUTE, each computed field must have its own ON MATCH COMPUTE statement.

The following request calculates employees' new salaries giving them a 10% increase over their present salaries. It only performs this calculations for employees whose IDs are stored in the data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
    CURR_SAL = D.CURR_SAL * 1.1;
  ON MATCH UPDATE CURR_SAL
DATA
```

Example Changing Incoming Data

You can use the COMPUTE statement to change incoming data. For example, assume you are preparing a MODIFY request to input new salaries into the data source. Just recently, the company granted employees in the MIS department an extra 3% pay raise. Rather than manually recalculating the new salaries for MIS employees, you can include a COMPUTE statement to do it for you:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL DEPARTMENT
COMPUTE
CURR_SAL = IF DEPARTMENT IS 'MIS'
    THEN CURR_SAL * 1.03
    ELSE CURR_SAL;
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
DATA
```

The new salary of employees who work in the MIS department will be 1.03 times more what they would have received ordinarily. Everybody else gets a normal raise.

Syntax How to Define Non-Data Source Transaction Fields

If the names of incoming data fields are not listed in the Master File describing the data source, you must define them to FOCUS before they are read in by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement. Otherwise, FOCUS rejects the fields as unidentifiable and terminates the request.

To define the fields to FOCUS, specify them with the COMPUTE statement using the notation

```
COMPUTE field/format=;
```

where:

field

Is the incoming data field you want to define to FOCUS.

format

Is the format of the field. Field formats are described in the *Describing Data* manual.

Because there is no expression after the equal sign (=), the request reads the statement before it reads the incoming data. All COMPUTE statements having expressions are executed after the request reads the incoming data.

For example, you want to record promotions to the MIS and Production Departments in the data source. However, the transaction data source you are working with lists the departments by code, not by name: a 1 for MIS and a 2 for Production. You prepare the following MODIFY request:

```

MODIFY FILE EMPLOYEE
COMPUTE DEPCODE/I1=;
PROMPT EMP_ID DEPCODE
COMPUTE
    DEPARTMENT = IF DEPCODE IS 1 THEN 'MIS' ELSE 'PRODUCTION';
MATCH EMP_ID
    ON MATCH UPDATE DEPARTMENT
    ON NOMATCH REJECT
DATA
    
```

The first COMPUTE statement defines the incoming DEPCODE field to FOCUS. The second COMPUTE statement sets the value of the transaction field DEPARTMENT depending on the value of DEPCODE. This DEPARTMENT field then updates the DEPARTMENT field in the data source.

Validating Transaction Values: The VALIDATE Statement

Most applications require that data be checked for accuracy before it is accepted into the data source. The VALIDATE statement checks values against certain conditions. If the value fails the test, the request rejects the transaction and displays a warning to the user.

For example, assume you are preparing a MODIFY request to update MIS and Production Department salaries in the data source. No one in those departments is ever paid less than \$6,000 per year or more than \$50,000. You can use the VALIDATE statement to reject those values that fall outside this range, such as a \$700 or a \$75,000 salary.

VALIDATE statements work the same way as COMPUTE statements: they set the value of a temporary field to the value of an expression. The only difference is that if the field value is set to 0, FOCUS rejects the transaction being processed and displays this message

```
(FOC421) TRANS n REJECTED INVALID rcode
```

where:

n

Is the number of the transaction being tested.

rcode

Is the variable receiving the test value.

The simplest way to use VALIDATE statements is to have them test the values of incoming data fields. If an incoming value is unacceptable, assign the temporary field a value of 0. Otherwise, assign the field a non-zero value. Note that the temporary field retains its value after the VALIDATE statement, and you may use this value in other calculations.

Tests provided by the DBA functions, which control access to data sources, function as involuntary VALIDATE tests and produce similar error messages.

You can place VALIDATE statements in two places in MODIFY requests:

- At the beginning of the request. VALIDATE statements here test every transaction, discarding those containing invalid values. Expressions in these VALIDATE statements cannot use data source field values (D. fields).
- In MATCH and NEXT statements. VALIDATE statements here test the transaction depending whether or not the MATCH or NEXT statement selected a particular segment instance. Expressions in these VALIDATE statements can use data source field values.

If you are validating fields in a repeating group and one field is rejected, all fields in the repeating group are rejected. However, if you are validating the fields in a MATCH or NEXT statement and one field is rejected, the other fields are not rejected.

If the MODIFY request prompts for data (the PROMPT statement), it is a good idea to validate each field after prompting. If you validate several fields at once, users must enter data for all the fields before the values they enter are tested. If one data value is invalid, they must reenter all the data values. If you validate each field, users will be warned as soon as they enter an invalid value, and the request will reprompt them for the correct value.

This section describes:

- VALIDATE statement syntax.
- Using the VALIDATE statement to validate incoming data.
- Use of the ON INVALID phrase.
- Use of VALIDATE statements in MATCH and NEXT statements.
- Testing for the presence of incoming data.
- Use of the DECODE function in VALIDATE statements.

If you validate data entered on a CRTFORM, invalid values cause the CRTFORM screen to be redisplayed along with the data you entered. This allows you to correct the data and re-enter it. You can deactivate this feature using the DEACTIVATE INVALID feature described in *Active and Inactive Fields* on page 9-199.

Syntax **How to Use a VALIDATE Statement**

The syntax of the VALIDATE statement is as follows (note that you may include several VALIDATE statements after the VALIDATE keyword)

```
VALIDATE
  field[/format] = expression;
  field[/format] = expression;
  .
  .
  .
```

where:

field

Is the name of the temporary field. If this field is set to 0, FOCUS rejects the transaction being processed. Do not use an incoming field name or data source field name for this name.

format

Is the format of the field. The format type must be numeric (I, F, D, or P. Formats are described in the *Describing Data* manual). You need to specify the format only if you will use the field elsewhere in the request.

expression;

Is any expression valid in a DEFINE or TABLE COMPUTE statement (See the *Creating Reports* manual.). Also, you may use the LOOKUP and FIND function described in *Special Functions* on page 9-107. If the value of the expression is 0, FOCUS rejects the transaction being processed. Note that you must terminate the expression with a semicolon (;).

You may specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

Reference Using VALIDATE to Test Incoming Data

You use VALIDATE statements most often to test incoming data values, assigning the temporary field a value of 0 if a value is not acceptable. The test expression can span several lines, but it must end with a semi-colon (;). Tests you can use in VALIDATE expressions are:

- IF...THEN...ELSE statements.
- Arithmetic expressions.
- Logical expressions.
- User functions and subroutines.
- DECODE functions.
- FIND and LOOKUP functions (see *Special Functions* on page 9-107).

You can use IF...THEN...ELSE statements in VALIDATE expressions (up to 16 statements per expression), such as:

```
SALTEST = IF SALARY LT 50000 THEN 1 ELSE 0;
```

If the incoming SALARY value is less than \$50,000, the SALTEST temporary field is set to 1. If SALARY is \$50,000 or greater, SALTEST is set to 0 and the transaction is rejected. Note that you may use all operations in VALIDATE IF...THEN...ELSE statements that you use in COMPUTE and DEFINE statements (see the *Creating Reports* manual). Also note that all alphanumeric literals must be enclosed in single quotation marks.

Example Using Logical Expressions

If an expression is evaluated as true, the temporary field is set to 1. Otherwise, the field is set to 0. For example:

```
SALTEST = SALARY LT 50000;
```

Note that you can use AND and OR operands in logical expressions, as discussed in the *Creating Reports* manual. For example:

```
SALTEST = (SALARY LT 50000) AND (JOB EQ 'B12');
```

If the incoming salary value is less than \$50,000 and the job code is B12, SALTEST is set to 1. Otherwise, SALTEST is set to 0.

Example Using the DECODE Function

This function allows you to compare an incoming field value against a list of acceptable and unacceptable values. For example:

```
SALTEST = DECODE JOBCODE (A03 0 B07 0 B12 0 ELSE 1);
```

If the incoming job code is A03, B07, or B12, SALTEST is set to 0.

Example Using the FIND Function

This function searches another FOCUS data source for the presence of the incoming field value. If the value is there, the temporary field is set to a non-zero value; otherwise the field is set to 0. For example:

```
SALTEST = FIND(EMP_ID IN EDUCFILE);
```

If the incoming employee ID value is not present in the EDUCFILE data source, SALTEST is set to 0. The FIND function is discussed in *Special Functions* on page 9-107.

The following MODIFY request validates the DEPARTMENT and CURR_SAL fields:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT CURR_SAL
VALIDATE
    DEPTEST = IF DEPARTMENT IS 'MIS' THEN 1 ELSE 0;
    SALTEST = CURR_SAL LT 50000;
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE CURR_SAL
DATA
```

This request will only accept your transactions if you enter MIS for the DEPARTMENT field and a value less than 50,000 for the CURR_SAL field.

Syntax **How to Take Action on Invalid Data: The ON INVALID Phrase**

If a VALIDATE statement invalidates a transaction, you may take action using the ON INVALID phrase. This phrase allows you to:

- Branch to another case using Case Logic. Case Logic is discussed in *Case Logic* on page 9-132.
- Type a message. Typing messages are discussed in *Messages: TYPE, LOG, and HELPMESSAGE* on page 9-117.

The ON INVALID phrase immediately follows the validate statement. The syntax is

```
ON INVALID GOTO casename
ON INVALID PERFORM casename
ON INVALID TYPE [ON ddname]
```

where:

```
GOTO casename
```

Branches to another case called *casename*. GOTO also takes other options described in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 9-137.

```
PERFORM casename
```

Branches to another case called *casename*. Execution then continues with the next statement after ON INVALID. PERFORM also takes other options discussed in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 9-137.

```
TYPE [ON ddname]
```

Displays a message of up to four lines on the terminal. If you use the ON *ddname* option, the request writes the message to a sequential data source allocated to *ddname*.

This request updates employee salaries. It warns you when you have entered a salary that fails its validation test:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
VALIDATE
    SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
    ON INVALID TYPE
        "YOU ENTERED A SALARY HIGHER THAN $50,000"
        "THIS SALARY IS TOO HIGH"
        "PLEASE REENTER THE EMPLOYEE ID AND SALARY"
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE CURR_SAL
DATA
```

VALIDATE Phrases in MATCH and NEXT Statements

You may place VALIDATE statements in MATCH and NEXT statements. The request only performs the validation if the MATCH or NEXT condition is met. These VALIDATE phrases may use data source fields if these fields are either in the segment instance being modified or in a parent instance along the segment path (the parent instance, the parent's parent, and so on until the root segment). To specify data source field values, affix the D. prefix to the front of the field name.

Note that VALIDATE statements that follow a MATCH or NEXT statement may also use data source fields if these fields are in the instance selected by the previous statement (or are in the segment path).

This request makes sure that an employee's new salary is not less than the present salary after it ascertains that the employee's ID is recorded in the data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT CURR_SAL
  ON MATCH VALIDATE
    SALTEST = IF CURR_SAL GE D.CURR_SAL THEN 1
              ELSE 0;
  ON MATCH UPDATE CURR_SAL
DATA
```

Example Testing for the Presence of Transaction Data

You may test for missing data values in transactions using the MISSING feature in IF and WHERE phrases, described in the *Creating Reports* manual. These features determine whether an incoming field is present in the transaction or not, and are especially useful when the transactions are in a transaction data source.

This request rejects transactions without a job code:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID CURR_JOBCODE CURR_SAL
VALIDATE
  JOBTTEST = IF CURR_JOBCODE IS NOT MISSING THEN 1
             ELSE 0;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_JOBCODE CURR_SAL
DATA
EMP_ID=071382660, CURR_JOBCODE=A13, CURR_SAL=18500.00, $
EMP_ID=112847612, CURR_SAL=19200.50, $
END
```

Syntax **How to Validate Values From a List: The DECODE Function**

The DECODE function allows you to compare incoming data values against a list of acceptable and unacceptable values. This function is described in the *Creating Reports* manual. This section discusses how best to use the DECODE function to validate data.

The syntax of the DECODE function is

```
field = DECODE fieldname (code1 result1...[ELSE default]);
```

where:

field

Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming field name or data source field name for this name.

fieldname

Is the incoming data field being tested.

code1 ...

Is the list of possible values.

result1

Is the number that the temporary field is set to if the incoming field has the preceding value. Place a 0 after invalid values; place a non-zero number after valid values.

ELSE

Indicates what the temporary field is set to if the incoming field does not have a value on the list. This list may have up to 32,767 literals.

For example, you want to record promotions to various company departments in the data source. There are five possible departments: Marketing, Accounting, Shipping, Sales, and Data Processing. You prepare this MODIFY request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT
VALIDATE
    DEPTTEST = DECODE DEPARTMENT (MARKETING 1
        ACCOUNTING 1 SHIPPING 1 SALES 1 MIS 1
        ELSE 0);
MATCH EMP_ID
    ON MATCH UPDATE DEPARTMENT
    ON NOMATCH REJECT
DATA
```

This request accepts MARKETING, ACCOUNTING, SHIPPING, SALES, and MIS as valid incoming values for the field DEPARTMENT, but rejects all other values.

You may also store the values in a separate file. The file must consist of stacked pairs of values, the values in each pair separated by a comma or spaces (you may want to arrange them in columns, see the example below). The left member of each pair is a possible value and the right member is the value that the temporary field is set to should the incoming data field have the value on the left.

The syntax of this form of the DECODE command is

```
field = DECODE infield (ddname ELSE m)
```

where:

field

Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming or data source field name for this name.

infield

Is the incoming field being tested.

ddname

Is the ddname of the file containing the list of possible values. The file may contain up to 32,767 bytes.

m

Is the value of *field* if the incoming data value is not in the list.

Below is a sample DECODE file.

```
MARKETING 1
ACCOUNTING 1
SHIPPING 1
SALES 1
MIS 1
```

Special Functions

There are two functions that you can use only in MODIFY COMPUTE and VALIDATE statements. They are:

- The FIND function, which tests for the existence of indexed values in FOCUS data sources.
- The LOOKUP function, which tests for the existence of non-indexed values in cross-referenced FOCUS data sources and makes these values available for other computations.

Note: The LAST function in MODIFY can be used in COMPUTEs and VALIDATEs, in combination with FREEFORM or FIXFORM, to test incoming transaction values against those from a previously read record. For further information on the LAST function see the *Creating Reports* manual.

Syntax **How to Test for the Existence of Indexed Values in FOCUS Data Sources: The FIND Function**

The FIND function verifies if an incoming data value is in a FOCUS data source field, whether the field is in the data source you are modifying or in another data source. The function sets a temporary field to a non-zero value if the incoming value is in the data source field and 0 if it is not. Note that a value greater than zero confirms the presence of the data value, not the number of instances in the data source field. You can then test and branch on this field using Case Logic, described in *Case Logic* on page 9-132.

Note that the data source field you are searching must be indexed, and that the FIND function does not work on data sources with different DBA passwords.

The syntax of the FIND function is

```
field = FIND(fieldname [AS dbfield] IN file);
```

where:

field

Is the name of the temporary field.

fieldname

Is the full name (not the alias or a truncation) of the incoming field being tested.

AS *dbfield*

Is the full name (not the alias or a truncation) of the data source field containing values to be compared with the incoming data field. This field must be indexed. If the incoming field and the data source field have the same name, you can omit this phrase.

file

Is the name of the data source.

Note that there can be no space between FIND and the left parenthesis.

The opposite of FIND is NOT FIND. The NOT FIND function sets a temporary field to 1 if the incoming value is not in the data source and 0 if the incoming value is in the data source. Its syntax is

```
field = NOT FIND(infield [AS dbfield] IN file)
```

where *field*, *infield*, *dbfield*, and *file* were explained previously.

You can use any number of FIND functions in COMPUTE and VALIDATE statements. However more FIND functions increase processing time and require more buffer space in core.

This request tests if each employee ID entered is also in the EDUCFILE data source. It then displays a message informing you whether it found the ID in the data source or not.

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID
COMPUTE
    EDTEST = FIND(EMP_ID IN EDUCFILE);
    MSG/A40 = IF EDTEST IS 1 THEN
        'STUDENT LISTED IN EDUCATION FILE' ELSE
        'STUDENT NOT LISTED IN EDUCATION FILE';
MATCH EMP_ID
    ON NOMATCH TYPE "<MSG"
    ON MATCH TYPE "<MSG"
DATA
    
```

Example Using the FIND Function in VALIDATE Statements

You may use the FIND function in a VALIDATE statement to test if a transaction field value exists in another FOCUS data source. If the field value is not in that data source, the function returns a value of 0, causing the validation to fail and the request to reject the transaction.

This request updates the number of hours spent by employees in class. It rejects employees not listed in the EDUCFILE data source, which records class attendance:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
    EDTEST = FIND(EMP_ID IN EDUCFILE);
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE ED_HRS
DATA
    
```

This VALIDATE statement will discard any incoming EMP_ID value not found in the EDUCFILE data source.

Reading Cross-Referenced FOCUS Data Sources: The LOOKUP Function

The LOOKUP function retrieves data values from cross-referenced data sources, both data sources cross-referenced statically in the Master File and data sources joined dynamically by the JOIN command. The LOOKUP function is necessary because, unlike TABLE requests, MODIFY requests cannot read cross-referenced data sources freely. With the LOOKUP function, the requests can use the data in computations and in messages but cannot modify cross-referenced data sources; to modify more than one data source in one request, use the COMBINE command discussed in *Modifying Multiple Data Sources in One Request: The COMBINE Command* on page 9-190.

The LOOKUP function can read cross-referenced segments that are linked directly to a segment in the host data source (the host segment). This means that the cross-referenced segments must have segment types of KU, KM, DKU, or DKM (but not KL or KLU) or contain the cross-referenced field specified by the JOIN command (see the *Describing Data* manual).

The cross-referenced segment contains two fields of interest:

- The field containing the values you want. This is the field the LOOKUP function specifies. For example, this LOOKUP function retrieves values from the DATE_ATTEND field:

```
RTN = LOOKUP (DATE_ATTEND) ;
```

- The cross-referenced field. This field shares values with a field in the host segment called the host field. These two fields link the host segment to the cross-referenced segment. The LOOKUP function uses the cross-referenced field, which is indexed, to locate a specific segment instance.

To use the LOOKUP function, the MODIFY request reads a transaction value for the host field. The LOOKUP function then searches the cross-referenced segment for an instance containing this value in the cross-referenced field:

- If there are no such instances, the function sets a return variable to 0. If you use the field specified by the LOOKUP function in the request, the field assumes a value of blank if alphanumeric and 0 if numeric.
- If there are instances (there can be more than one if the cross-referenced segment type is KM, DKM, or if you specified the ALL keyword in the JOIN command), the function sets the return variable to one and retrieves the value of the specified field from the first instance it finds.

The syntax of the LOOKUP function is

```
rcode = LOOKUP(field);
```

where:

*r*code

Is a variable you specify to receive a return code value. This value is 1 if the LOOKUP function can locate a cross-referenced segment instance, 0 if the function cannot.

field

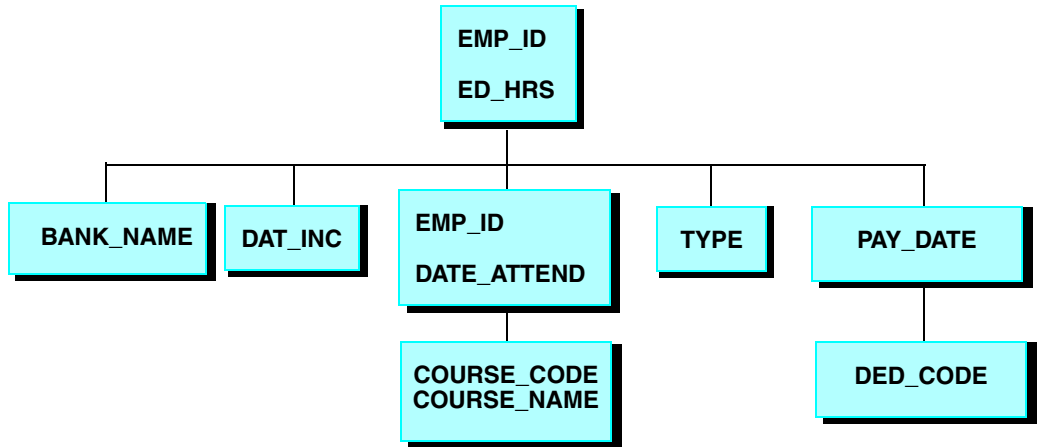
Is the field that you want to retrieve in the cross-referenced data source. Note that this field name cannot exist in the host data source, and that the LOOKUP function may specify only one field at a time. Each field you wish to retrieve requires a separate LOOKUP function. To look up all fields in the cross-referenced segment, use LOOKUP (SEG.field).

Note that there may be no space between LOOKUP and the left parenthesis. The LOOKUP function can exist by itself or as part of a larger expression. If it exists by itself, it must terminate with a semicolon.

For example, you wish to update the amount of classroom hours employees have spent. Because of a new system of accounting, employees taking classes after January 1, 1985 are to be credited with 10% more classroom hours than their records indicate.

The employee IDs (EMP_ID) and classroom hours (ED_HRS) are located in the host segment. The class dates (DATE_ATTEND) are located in the cross-referenced segment. The shared field is the employee ID field.

The data source structure is shown in this diagram:



The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
COMPUTE
  EDTEST = LOOKUP (DATE_ATTEND) ;
  COMPUTE
  ED_HRS = IF DATE_ATTEND GE 820101 THEN ED_HRS * 1.1
          ELSE ED_HRS ;
MATCH EMP_ID
  ON MATCH UPDATE ED_HRS
  ON NOMATCH REJECT
DATA
```

A sample execution of this request might go as follows:

1. The request prompts you for an employee ID and number of class hours. You enter the ID 117593129 and 10 class hours.
2. The LOOKUP function locates the first instance in the cross-referenced segment containing the employee ID 117593129. Since the instance exists, the function returns a 1 to the EDTEST variable. This instance lists the class date as 821028 (October 28, 1982).
3. The LOOKUP function retrieves the value 821028 for the DATE_ATTEND field.
4. The COMPUTE statement tests the value of the DATE_ATTEND field. Since October 28, 1982 is after January 1, 1982, the statement increases the incoming ED_HRS value from 10 to 11 hours.
5. The request updates the classroom hours for employee 117593129 using the new ED_HRS value.

You may also use a data source value in a specific host segment instance to search the cross-referenced segment. To do this, prepare the request this way:

- In the MATCH statement that selects the host segment instance, activate the host field. This can be done with the ACTIVATE phrase (discussed in *Active and Inactive Fields* on page 9-199).
- In the same MATCH statement, place the LOOKUP function after the ACTIVATE phrase.

This request displays the employee IDs, dates of salary raises, employee names, and the position each employee held after the raise was granted:

- The employee IDs and names (EMP_ID) are in the root segment.
- The date of raise (DAT_INC) is in the descendant host segment.
- The job titles are in the cross-referenced segment.
- The shared field is JOBCODE. You never enter any job codes; the values are all stored in the data source.

The request is:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID DAT_INC
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DAT_INC
  ON NOMATCH REJECT
  ON MATCH ACTIVATE JOBCODE
  ON MATCH COMPUTE
    RTN = LOOKUP (JOB_DESC) ;
  ON MATCH TYPE
    "EMPLOYEE ID:           <EMP_ID"
    "DATE INCREASE:        <DAT_INC"
    "NAME:   <D.FIRST_NAME  <D.LAST_NAME"
    "POSITION:             <JOB_DESC"
DATA

```

A sample execution might go as follows:

1. The request prompts you for an employee ID and date of pay raise. You enter employee ID 071382660 and date of raise 820101 (January 1, 1982).
2. The request locates the instance containing the ID 071382660, then locates the child instance containing the date of raise 820101.
3. This child instance contains the job code A07. The ACTIVATE statement activates this value, making it available to the LOOKUP function.

4. The LOOKUP function locates the job code A07 in the cross-referenced segment. It returns a 1 into the RTN variable and retrieves the corresponding job description of SECRETARY.
5. The request displays the values using a TYPE statement:

```
EMPLOYEE ID:      071382660
DATE INCREASE:   82/01/01
NAME:            ALFRED STEVENS
POSITION:        SECRETARY
```

Note: You may also need to activate the host field if you are using the LOOKUP function within a NEXT statement. This request, similar to the previous one except for the NEXT statement, displays the latest position held by a particular employee.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
NEXT DAT_INC
  ON NONEXT REJECT
  ON NEXT ACTIVATE JOBCODE
  ON NEXT COMPUTE
  RTN = LOOKUP (JOB_DESC) ;
  ON MATCH TYPE
    "EMPLOYEE ID:          <EMP_ID"
    "DATE OF POSITION:      <DAT_INC"
    "NAME: <D.FIRST_NAME  <D.LAST_NAME"
    "POSITION:            <JOB_DESC"
DATA
```

Syntax **How to Use an Extended Syntax With LOOKUP**

If the function cannot locate a value of the host field in the cross-referenced segment, you may specify that the LOOKUP function locate the next highest or lowest cross-referenced field value in the cross-referenced segment by using an extended syntax.

To use this LOOKUP feature, the index must have been created on FOCUS Release 4.5 or later with the INDEX parameter set to NEW (the binary tree scheme). To determine what type of index your data source uses, enter the ? FDT command (see the *Developing Applications* manual).

Note that a field retrieved by the LOOKUP function does not require the D. prefix to be displayed in TYPE statements. FOCUS treats the field value as a transaction value.

The extended syntax of the LOOKUP function is

```
COMPUTE
  rcode = LOOKUP(field operator);
```

where:

rcode

Is a variable you specify to receive a return code value. (The value the variable receives depends on the outcome of the function below.)

field

Is the name of the field you want to use in MODIFY computations. Note that this cannot be the cross-referenced field.

operator

These parameters specify the action the request takes if there is no cross-referenced segment instance corresponding to the host field value. The actions can be one of the following:

EQ causes the LOOKUP function to take no further action if an exact match is not found. If a match is found, the value of *rcode* is set to 1; otherwise, it is set to 0. This is the default.

GE causes the LOOKUP function to locate the instance with the exact or next highest value of the cross-referenced field.

LE causes the LOOKUP function to locate the instance with the exact or next lowest value of the indexed field.

Note that there can be no space between LOOKUP and the left parenthesis.

This table summarizes the value of *r*code depending on which instance the LOOKUP function locates:

Action	rcode value
Exact cross-referenced value located	1
Next highest cross-referenced value located	2
Next lowest cross-referenced value located	-2
Cross-referenced field value not located	0

Reference Using the LOOKUP Function in VALIDATE Statements

When you use the LOOKUP function, you may want to reject transactions containing values for which there is no corresponding instance in the cross-reference segment. To do this, place the function in a VALIDATE statement. If the function cannot locate the instance in the cross-referenced segment, it sets the value of the return variable to 0. This causes the request to reject the transaction.

The following request updates an employee's classroom hours (ED_HRS). If the employee attended classes on or after January 1, 1982, the request increases the number of classroom hours by 10%. The classroom attendance dates are stored in a cross-referenced segment (field DATE_ATTEND). The shared field is the employee ID.

The request is:

```

MODIFY FIELD EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
    TEST_DATE = LOOKUP (DATE_ATTEND) ;
COMPUTE
    ED_HRS = IF DATE_ATTEND GE 820101 THEN ED_HRS * 1.1
            ELSE ED_HRS;
MATCH EMP_ID
    ON MATCH UPDATE ED_HRS
    ON NOMATCH REJECT
DATA
    
```

If the employee is not recorded in the cross-referenced segment, then the employee has never attended a class. This means that a transaction recording the employee's classroom hours is an error, and should be rejected.

This is the purpose of the LOOKUP function in the VALIDATE statement. If the function cannot locate an employee's record in the cross-referenced segment, it returns a 0 to the TEST_DATE field. This causes the request to reject the transaction.

Messages: TYPE, LOG, and HELPMESSAGE

This section describes how MODIFY requests handle messages. There are four types:

- Messages written into requests.
- Messages indicating the rejection of transactions.
- Messages originating from the Master File with the HELPMESSAGE attribute.
- Messages that echo transactions.

These messages are helpful in debugging MODIFY requests, locating rejected transactions, and instructing the operator. There are two statements and one attribute that control the display of messages:

- The TYPE statement enables you to write messages to the terminal and to sequential files.
- The LOG statement stores incoming or rejected transactions in sequential files and controls the display of rejection messages.
- The HELPMESSAGE attribute is a field attribute included in the Master File (of FOCUS data sources). Text messages specified in the Master are displayed in the TYPE area of MODIFY CRTFORMs.

Displaying Specific Messages: The TYPE Statement

The TYPE statement either appears on the terminal or stores in a sequential file messages that you prepare. This section describes:

- The syntax of the TYPE statement.
- Use of embedded data fields.
- Use of spot markers.
- Use of extended attributes.

Note: Text fields cannot be used with the TYPE statement.

Syntax **How to Use a TYPE Statement**

The syntax of the TYPE statement is

```
TYPE [AT START|AT END] [ON ddname]
"message"
["message"]
```

where:

AT START

Displays a message at the beginning of execution only.

AT END

Displays a message at the end of execution only. If you are using Case Logic, the TYPE AT END statement must be in the case that generates the end-of-file condition. Either the case must include a FIXFORM or FREEFORM statement that will reach the end of the transaction data source; or a PROMPT statement, at which the user will type END or QUIT; or a CRTFORM statement, at which the user will type END or press the PF3 key.

ON ddname

Writes the message to a sequential file allocated to *ddname*. The TYPE statement can write lines of up to 256 characters each, including blanks and embedded field values. If you omit this phrase, the request displays the message on the terminal.

message

Is any message. Enclose each line in double quotation marks (except when you want to display two lines as one line, as described later in this section in *Embedding Spot Markers* on page 9-122.) If you are displaying messages at the terminal, the lines begin in column 2 on the screen. If you are writing the message to a file, the lines begin in column 3 in the file. You may embed spot markers and data fields in the message.

Note that you can type the TYPE statement on one line. For example:

```
TYPE "THIS IS A ONE LINE MESSAGE"
```

TYPE statements can stand by themselves, they can be part of MATCH and NEXT statements, and they can follow VALIDATE statements. For example:

```
MODIFY FILE EMPLOYEE
TYPE
  " "
  "PLEASE ENTER THE FOLLOWING DATA"
  " "
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
```


This request asks the user to enter data at the beginning of every transaction. Note that there is a blank message line both before and after the message "PLEASE ENTER THE FOLLOWING DATA:" This enhances readability and appearance.

TYPE statements may be part of MATCH and NEXT statements. For example, this request warns the user when an employee ID that the user has entered is not in the data source:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
    " "
    "NO SUCH EMPLOYEE IN THE DATABASE"
    "PLEASE RETYPE THE EMPLOYEE ID"
  ON NOMATCH REJECT
DATA

```

TYPE statements can display messages when incoming data values fail validation tests, as discussed in *Validating Transaction Values: The VALIDATE Statement* on page 9-99. For example, this request warns the user when a salary higher than \$50,000 is entered:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
VALIDATE
  SALTEST = IF CURR_SAL LE 50000 THEN 1 ELSE 0;
  ON INVALID TYPE
    " "
    "THE CURR_SAL VALUE IS OVER 50000"
    "AND THEREFORE CANNOT BE ENTERED INTO THE"
    "DATABASE. PLEASE NOTIFY YOUR SUPERVISOR."
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA

```

Note that ON INVALID TYPE phrases can occur after VALIDATE statements that stand by themselves or are part of MATCH statements. For example:

```

MATCH PAY_DATE
ON NOMATCH REJECT
ON MATCH VALIDATE
  GROSS_TEST = IF GROSS LT 1500 THEN 1 ELSE 0;
  ON INVALID TYPE
    "GROSS OVER $1500. PLEASE REENTER"

```

Reference Embedding Data Fields

You can embed data fields in the middle of messages. Embedded data fields are described in the *Creating Reports* manual. The kind of field you may embed depends on the position of the TYPE statement:

- TYPE statements preceding MATCH or NEXT statements only accept incoming data fields in messages, not data source fields.
- This request contains a TYPE statement before the MATCH statement:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 CURR_SAL/8
TYPE
    "EMPLOYEE ID: <EMP_ID SALARY: <CURR_SAL"
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
DATA ON EMPSAL
END
```

- TYPE phrases in or following a MATCH or NEXT statement accept both incoming data fields and data source fields in messages. The data source field must either be in the segment instance that the MATCH or NEXT statement is modifying or in a parent instance along the segment path (the parent instance, the parent's parent, and so on to the root segment). To specify a data source field, affix the prefix D. to the field name.

This TYPE phrase displays both the incoming value of CURR_SAL and the data source value:

```
ON MATCH TYPE
    "SALARY ENTERED IS: <CURR_SAL"
    "OLD SALARY WAS: <D.CURR_SAL"
```

You can use embedded fields together in a statement to display a total. This request totals all salaries updated:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH COMPUTE
        TOTAL_SAL/D10.2 = TOTAL_SAL + CURR_SAL;
    ON MATCH UPDATE CURR_SAL
TYPE AT END
    "TOTAL OF ALL NEW SALARIES IS <TOTAL_SAL"
DATA
```

Every time the user enters a salary, the request adds it to the running total TOTAL_SAL. After the user enters the last salary, the request displays the TOTAL_SAL value embedded in the message.

Note: Each line of text can contain up to 256 characters. This includes the lengths of the embedded fields as defined by the display field formats (for example, the CURR_SAL field, having the format D12.2M, takes up 15 characters, including decimal point, commas, and dollar sign).

Embedded fields enable you to design your own log files to record transactions, replacing the automatic log file facility activated by the LOG statement. This request logs accepted transactions into the file ACCFILE and logs rejected transactions into the file REJFILE:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH TYPE ON ACCFILE
    "<EMP_ID <12 <CURR_SAL"
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE ON REJFILE
    "<EMP_ID <12 <CURR_SAL"
  ON NOMATCH REJECT
DATA
```

This request records in the ACCFILE file the employee ID and new salary entered by the user if the ID is in the data source and records the ID and salary in the REJFILE file if the ID is not in the data source. Note that the spot markers in both TYPE messages ensure that the fields will be aligned in the files, making the files fixed sequential files. If the request logged the transactions using the MODIFY LOG facility, the files would have been comma-delimited because the request uses PROMPT to input data. Note that you must issue a FILEDEF or allocation for each log file prior to using it in the MODIFY request.

Reference Embedding Spot Markers

You can embed spot markers in TYPE statement messages. Spot markers are devices that place message text at different places on the screen. Spot markers are described in Chapter 4, *Tutorial: Painting a Procedure*. Some common spot markers are shown below (where *n* is an integer):

<n

Places text starting at the *n*th column.

<+n

Places text *n* columns to the right.

</n

Places text *n* lines down.

<0X

Positions the next character immediately to the right of the last character (skip zero columns). This is used when you have two or more lines between the double quotation marks in a procedure that make up a single line of information on a FIDEL screen. No spaces are inserted between the spot marker and the start of a continuation line.

For example, the statement

```
TYPE
  "THE DOLLAR SIGN IS IN COLUMN 40: <40 $"
  "TEN SPACES ARE EMBEDDED <+10 IN THIS LINE"
  "</1 THIS LINE SKIPS A LINE <0X
  AND PROVIDES AN EXAMPLE OF THE USE <0X
  OF A COLUMN MARKER"
```

produces the following output:

```
THE DOLLAR SIGN IS IN COLUMN 40:      $
TEN SPACES ARE EMBEDDED                IN THIS LINE

THIS LINE SKIPS A LINE AND PROVIDES AN EXAMPLE OF THE USE OF A COLUMN MARKER
```

Note: The spot marker to skip a line, </n, can appear on the same line with other text in a TYPE statement. However, in a CRTFORM, this spot marker must appear on a line by itself (see Chapter 10, *Designing Screens With FIDEL*).

Sometimes, a line of text you want displayed cannot fit on one line within the TYPE command. This can occur because you are indenting lines or because there are non-printable characters in the message, such as spot markers and field prefixes. To have two lines in the TYPE statement displayed as one line, do the following:

- End the first line without an end quotation mark.
- Do not begin the second line with a quotation mark. Instead, begin the line with a <+n spot marker where *n* is any number greater than or equal to zero.

This TYPE statement demonstrates how this feature can be used:

```
TYPE
  "<D.FIRST_NAME <D.LAST_NAME EMP. #<EMP_ID
  <+1 SALARY: <CURR_SAL"
```

If you enter in the employee ID 123764317 and a salary of \$27,000, the request displays this message:

```
JOAN IRVING      EMP. #123764317 SALARY: $27,000.00
```

You may write a message of several lines this way. Begin the first line of the message with a quotation mark and end the last line with a quotation mark. Begin alternating lines with the <+1 spot marker. This causes the request to display every two lines of text as one line.

For example, if you type this statement in the request:

```
TYPE
  "SALARY UPDATE PROCEDURE
  <+1 WRITTEN JUNE 26, 1985"
  "ENTER EACH EMPLOYEE ID AND SALARY
  <+1 AFTER THE PROMPTS"
```

The request displays the message as:

```
SALARY UPDATE PROCEDURE WRITTEN JUNE 26, 1985
ENTER EACH EMPLOYEE ID AND SALARY AFTER THE PROMPTS
```

The following request employs both spot markers and embedded fields in messages:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH TYPE
    "</1 EMPLOYEE <EMP_ID NOT IN THE DATABASE"
    "PLEASE RETYPE NUMBER OR NOTIFY SUPERVISOR"
  ON NOMATCH REJECT
  ON MATCH TYPE
    "</1 EMPLOYEE <15 LAST_NAME <30 FIRST_NAME <45 SALARY"
    "</1 <EMP_ID <15 <D.LAST_NAME
    "<+1 <30 <D.FIRST_NAME <40 <D.CURR_SAL"
    "</1 ENTER SALARY FOR EMPLOYEE <EMP_ID"
    " "
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
DATA
```

When you run this request, the session looks like this:

```
> EMPLOYEE ON 10/10/98 AT 19.44.47
DATA FOR TRANSACTION 1

EMP_ID      = > 451123478

EMPLOYEE      LAST_NAME  FIRST_NAME  SALARY
451123478     MCKNIGHT  ROGER      $16,100.00

ENTER SALARY FOR EMPLOYEE 451123478

CURR_SAL     = > 18500
DATA FOR TRANSACTION 2

EMP_ID      = >
```

Reference Screen Attributes

If your request includes CRTFORMs, you can enhance TYPE statements with screen attributes, devices that display a line, part of a line, or an embedded field in color, in reverse video, flashing, or underlined. Screen attributes are discussed in Chapter 10, *Designing Screens With FIDEL*, in connection with the FIDEL facility.

Note the following when using screen attributes in TYPE statements:

- You may use screen attributes only in TYPE statements that follow a CRTFORM and will appear on the screen beneath the CRTFORM during execution.
- Extended attributes in TYPE statements only work on terminals that can process all screen attributes. To use screen attributes in TYPE statements, you must issue the command:

```
SET EXTTERM = ON
```

- When you add an attribute to a line, whether you place the attribute before a field or before text, the attribute remains in effect until the end of the line or until the next attribute, whichever comes first.
- Attributes for TYPE statements are cleared at the end of each line. To apply an attribute to a block of text, type the attribute at the beginning of each line.

This request uses attributes in TYPE statements:

```
MODIFY FILE EMPLOYEE
CRTFORM
"ENTER EMPLOYEE ID:      <EMP_ID"
"ENTER SALARY:          <CURR_SAL"

MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
    "<.WHITE. EMPLOYEE #<.AQUA.EMP_ID"
    "<.WHITE. IS <.RED. NOT <.WHITE. IN THE DATABASE"
    "<.WHITE. PLEASE NOTIFY SUPERVISOR"
  ON NOMATCH REJECT
DATA
END
```

The request displays the employee ID in aquamarine and the EMPLOYEE IS NOT IN THE DATABASE message in white, except for the word NOT, which is in red.

Logging Transactions: The LOG Statement

The LOG statement enables you to record transactions in sequential files automatically and to control the display of rejection messages at the terminal. You may use the LOG statement to record transactions in files, one file for each type of transaction: all transactions, accepted transactions, and different types of rejected transactions. The statement can also shut off MODIFY command rejection messages, enabling you to substitute your own.

Syntax How to Log Transactions in Sequential Files

The LOG statement enables you to record transactions processed by a MODIFY request in sequential files. You can record all transactions or only transactions accepted into the data source. You can record in separate files transactions rejected because of an ON MATCH REJECT or ON NOMATCH REJECT phrase, transactions rejected because of validation tests, and transactions rejected because of format errors.

Note that you can design your own log files by using the TYPE ON ddname statement described in *Displaying Specific Messages: The TYPE Statement* on page 9-117 instead of the LOG facility.

You add a LOG statement for each file in which you are storing transactions. The syntax for the LOG statement is

```
LOG category [ON ddname] [MSG {ON|OFF}]
```

where:

category

Is the type of transaction to be logged. The types are:

TRANS are all transactions processed by the request.

ACCEPT are transactions accepted into the data source.

DUPL are transactions rejected because of an ON MATCH REJECT phrase (the transactions have field values that match those in the data source).

NOMATCH are transactions rejected because of an ON NOMATCH REJECT phrase (the transactions have field values that do not match values in the data source).

INVALID are transactions rejected because of data values that failed validation tests.

FORMAT are transactions rejected because of data values that have invalid formats (for example: a numeric field containing letters; an alphanumeric field with more characters than allowed by the format). Any non-CRTFORM transaction that fails an ACCEPT test can also be logged to this file.

ddname

The ddname of the file to which you are writing.

MSG

Controls the display of rejection messages (messages displayed on the terminal when a transaction is rejected). The default setting is ON, except for ACCEPT where the default is OFF. The ON setting enables the display of rejection messages.

You can log messages on six files in one request. If the files existed before the user executed the request, the logged transactions replace the file contents.

How the request stores transactions depends on the statement used to read them in.

FIXFORM	The request stores the transactions in fixed format. Each FIXFORM statement retrieving data from the data source logs one transaction. Each transaction consists of the fields defined by the FIXFORM statement plus the fields to the end of the physical record.
FREEFORM	The request stores the transactions in comma-delimited format. Each FREEFORM statement logs one transaction. Each transaction consists of one physical record delimited by a comma-dollar sign (,\$). Note: Unless FREEFORM is explicitly included in the syntax, only the last line entered will be logged.
PROMPT	The request stores the transactions in comma-delimited format. Each PROMPT statement logs one transaction. Each transaction consists of data collected from the first PROMPT statement in the request to the PROMPT statement logging the transaction.
CRTFORM	The request stores the transactions in fixed format. Each CRTFORM logs one transaction. Each transaction consists of data collected from the first CRTFORM in the request to the CRTFORM logging the transaction.

When you allocate the files, you must assign each file a record length just large enough to hold the transaction. How you determine the length depends on how the request reads transactions:

<p>FIXFORM and FREEFORM</p>	<p>Define the record length as the length of the longest logical transaction record, including blanks and commas between the fields. Remember that a logical transaction record can extend over more than one line in the transaction data source (but is recorded as one line in the log file).</p>
<p>PROMPT</p>	<p>Define the record length as the sum of the lengths of the fields as defined by the FORMAT attribute (for example, a field having a format of D12.2 has a length of 12), plus one byte for each field, plus one more byte.</p>
<p>CRTFORM</p>	<p>Define the record length as the sum of the lengths of the fields as defined by the FORMAT attribute (for example, a field having a format of D12.2 has a length of 12), plus one byte for each CRTFORM, plus one more byte.</p>

The sample request below updates employee salaries and logs the transactions on five separate files. The original transaction data source was stored in file ddname SALFILE. Note the VALIDATE statement that determines whether the salary in each transaction exceeds \$50,000.

```

MODIFY FILE EMPLOYEE

LOG TRANS      ON ALLTRANS
LOG ACCEPT     ON GOODTRAN
LOG NOMATCH    ON NOEMPL
LOG INVALID    ON BIGSAL
LOG FORMAT     ON BADFORM

PROMPT EMP_ID CURR_SAL
VALIDATE
    SAL_TEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
DATA
    
```

Note the five files specified in the LOG statements:

- The ALLTRANS file records all transactions.
- The GOODTRAN file records transactions accepted into the data source.
- The NOEMPL file records transactions with employee IDs that do not exist in the data source.
- The BIGSAL file records transactions with salaries that are too big (over \$50,000).
- The BADFORM file records transactions with salaries having invalid characters.

Syntax **How to Control the Printing of Rejection Messages**

The MSG option on a LOG statement allows you to control the display of FOCUS automatic rejection messages. You can replace these messages by shutting them off and displaying your own messages using the TYPE command. The FOCUS messages are the following:

- For transactions rejected because of an ON MATCH REJECT phrase (the transactions have values that match values in the data source)

```
(FOC405 )TRANS n REJECTED DUPL: segment
```

where *n* is the transaction number and *segment* is the data source segment containing the data value that matched the transaction value.

- For transactions rejected because of an ON NOMATCH REJECT phrase (the transactions have values that do not match values in the data source)

```
(FOC415) TRANS n REJECTED NOMATCH segment
```

where *n* is the transaction number and *segment* is the data source segment containing the data field that failed to match the transaction value.

- For transactions rejected because of values that failed validation tests

```
(FOC421)TRANS n REJECTED INVALID field
```

where *n* is the transaction number and *field* is the return code field.

- For transactions read in using FIXFORM that were rejected because of values with format errors or ACCEPT errors

```
(FOC428)TRANS n REJECTED FORMAT COL m FLD field
```

where *n* is the transaction number, *m* is the first column of the field having the error, and *field* is the data field containing the error.

- For transactions read in using FREEFORM and PROMPT that were rejected because of values with format errors

```
(FOC210) THE DATA VALUE HAS A FORMAT ERROR: v
```

where *v* is the data value.

- For transactions read in using CRTFORM that were rejected because of values with format errors

```
SCREEN REJECTED.. FORMAT ERROR IN FIELD field
```

where *field* is the data field with the format error.

- For transactions read in using CRTFORM or PROMPT that were rejected because a value failed in an ACCEPT test

```
(FOC534) Data Value is not among the acceptable values for: field
```

where *field* is the data field containing the error.

In addition, FOCUS displays the rejected transaction after each rejection message (except for format error transactions read in using PROMPT and CRTFORM).

You may want to replace these messages with your own. To do so, use the TYPE statement described in *Displaying Specific Messages: The TYPE Statement* on page 9-117. To turn off the FOCUS messages, use the LOG statement with this syntax

```
LOG category [ON ddname] MSG {ON|OFF}
```

where:

category

Is the type of transaction that triggers the rejection message: DUPL, NOMATCH, INVALID, and FORMAT. These types are described previously in *How to Log Transactions in Sequential Files* on page 9-126.

ON *ddname*

Logs the transaction in a file defined by *ddname*. This option is described previously in *How to Log Transactions in Sequential Files* on page 9-126.

MSG

Is the parameter that turns FOCUS rejection messages ON (the default) or OFF.

For example, this request shuts off the automatic NOMATCH message and replaces it with another message:

```
MODIFY FILE EMPLOYEE
LOG NOMATCH MSG OFF
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
    "THIS EMPLOYEE IS NOT RECORDED IN THE DATABASE"
    "DID YOU ENTER THE ID NUMBER CORRECTLY?"
    "THE NUMBER YOU ENTERED WAS: <EMP_ID"
  ON NOMATCH REJECT
DATA
```

Note that you may combine logging and the display of rejection messages in one LOG statement. For example, to both log transactions rejected because of the ON NOMATCH REJECT phrase and shut off the FOCUS message that results from those transactions, you can use this LOG statement:

```
LOG NOMATCH ON NOEMPL MSG OFF
```

Adding the logging facility enables the end user to deal with problem transactions after entering all the data.

Displaying Messages: The HELPMESSAGE Attribute

The HELPMESSAGE attribute enables you to specify a text message in the Master File of FOCUS data sources. The message is displayed in the TYPE area of MODIFY CRTFORMs.

Syntax How to Specify a HELPMESSAGE Attribute

The syntax for specifying the HELPMESSAGE attribute in the Master File is

```
FIELDNAME=name, ALIAS=alias, FORMAT=format,
  HELPMESSAGE= text..., $
```

where:

text

Is a one-line text message up to 78 characters, which may include all characters and digits. Text containing a comma must be enclosed in single quotation marks; leading blanks are ignored.

For example:

```
FIELDNAME=LAST_NAME, ALIAS=LN, FORMAT=A10,
ACCEPT = SMITH JONES,
HELMESSAGE = 'LAST_NAME MUST BE SMITH, OR JONES', $
```

The field for LAST_NAME has an ACCEPT attribute that tests values entered for that field. If a value other than Smith or Jones is entered, the following messages will be displayed:

```
(FOC534) DATA VALUE IS NOT AMONG ACCEPTABLE VALUES FOR LAST_NAME
LAST_NAME MUST BE SMITH, OR JONES
```

The HELPMESSAGE attribute can be used with a field that has an ACCEPT test (see the *Describing Data* manual), or any other field in the Master File.

Messages specified with the HELPMESSAGE attribute are displayed when:

- The value entered for a data source field is invalid according to the ACCEPT test for that field.
- The value entered for a data source field causes a format error.
- The user places the cursor in the data entry area for a particular field and presses a predefined PF key.

Regardless of the condition that triggers display of the message specified with the HELPMESSAGE attribute, the same message will appear.

Displaying Messages: Setting PF Keys to HELP

In order to see the HELPMESSAGE text for a field on the CRTFORM, set a PF key to HELP before executing the MODIFY procedure. To set a PF key, enter

```
SET PFnn = HELP
```

where:

nn

Is the number of the PF key you want to define as your HELP key.

To display a message for a particular field, position the cursor on the data entry area for that field on the CRTFORM and press the defined PF Key. If no message has been specified for the field, the following message will be displayed:

```
NO HELP AVAILABLE FOR THIS FIELD.
```

Case Logic

Case Logic allows you to branch to different parts of MODIFY requests during execution. This enables you to construct more complex MODIFY requests. For example, Case Logic requests can offer the terminal operator the choice of different procedures, process different transaction records differently, or update multiple segment instances with a single transaction.

Case Logic also extends the use of the NEXT statement to process segment chains and facilitates modifying multiple unique child segments.

To prepare a request using Case Logic, you divide the request into sections called cases. Each case is labeled, allowing you to branch to the case from elsewhere in the request.

Syntax **How to Use a Case Statement**

Each case begins with the statement

```
CASE     {AT START|casename}
```

where:

AT START

Indicates that the case is to be executed only at the beginning of the request. This case is called the START case.

casename

Is a label of up to 12 characters that does not contain embedded blanks or the characters:

```
+ - * / & $ ' "
```

Each case ends with the statement:

```
ENDCASE
```

The CASE and ENDCASE statements must both be on lines by themselves.

The first case in the request, the one immediately following the MODIFY command, needs neither a beginning nor an ending statement. It is automatically assigned the label TOP. Note, however, that if the request contains only one case, you may want to begin the case with the statement CASE TOP and end it with ENDCASE. This allows you to branch to the beginning of the request from its middle.

The following request updates employee salaries in the EMPLOYEE data source. If the salary is above \$50,000, the request has the user retype the value to confirm it:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
IF CURR_SAL GT 50000 GOTO CONFIRM ELSE GOTO NEWSAL;

CASE NEWSAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE

CASE CONFIRM
TYPE
  "THE SALARY YOU ENTERED EXCEEDS $50,000"
  "PLEASE REENTER THE SALARY TO CONFIRM IT"
  "OR ENTER A NEW SALARY"
PROMPT CURR_SAL
GOTO NEWSAL
ENDCASE
DATA
```

This request consists of three cases: the TOP case, the NEWSAL case, and the CONFIRM case. (The blank lines between cases are there to enhance readability and are not required.)

The TOP case contains the first two statements in the request:

```
PROMPT EMP_ID CURR_SAL  
IF CURR_SAL GT 50000 GOTO CONFIRM
```

The TOP case prompts you for an employee ID and new salary. It then tests the salary value you entered. If the salary is more than \$50,000, it branches to the CONFIRM case. Otherwise, the request proceeds with the next case.

The next case is the NEWSAL case. This case updates the employee salaries. After the update, the request automatically returns to the beginning of the TOP case to prompt for the next employee ID and salary.

The third case is the CONFIRM case. This is where the request branches if you enter a salary higher than \$50,000. The case asks you to reenter the salary. It then branches to the NEWSAL case to enter the salary into the data source.

This is the order of cases executed if you enter a salary lower than \$50,000:

1. The TOP case.
2. The NEWSAL case.
3. Back to the TOP case.

This is the order of cases executed if you enter a salary higher than \$50,000:

1. The TOP case.
2. The CONFIRM case.
3. The NEWSAL case.
4. Back to the TOP case.

Rules Governing Cases

The following rules apply to cases:

- Each case (except for the TOP case) must begin with a CASE statement and end with an ENDCASE statement; both statements must appear on separate lines.
- Each case must have a unique name within the MODIFY request.
- The TOP case is always the first case in the procedure. It has no beginning or ending case statements. No other case may be labeled TOP.
- If the TOP case has both CRTFORM and COMPUTE commands, the CRTFORM (data entry) is processed before the computation.
- There can be only one START case. If you include a START case, it must come after the TOP case. The START case is discussed in *Executing a Case at the Beginning of a Request Only: The START Case* on page 9-137.
- No case may be named EXIT. The label EXIT refers to the end of the request.
- Except for the TOP case, which must come first, and the START case, which follows after, the cases may appear in the request in any order.
- Except for the TOP and START cases, you can execute a case only by using a GOTO, PERFORM, or IF statement to branch to it.
- At the end of a case, the request branches back to the TOP case unless a GOTO or IF statement states otherwise.
- You cannot branch to the middle of a case, only to its beginning.

Each case must contain complete MODIFY statements, not phrases or fragments. For example, the following case is illegal because ON NOMATCH REJECT is a phrase belonging to the MATCH statement.

```
CASE REJECT
ON NOMATCH REJECT
ENDCASE
```

- Cases cannot be nested; that is, you cannot put a case within another case. Each case must end before another can begin.
- You cannot have a statement between two cases except for comments. As soon as one case ends, the next case must begin.

- Certain MODIFY statements are global and apply to the request as a whole. We recommend that these statements follow the last case:

START
STOP
LOG
DATA
CHECK

- Cases do not allow you to use either the FREEFORM or the PROMPT statement in requests with FIXFORM or CRTFORM statements. You also cannot use more than one FIXFORM statement with CRTFORMs. For using FIXFORM statements with CRTFORMs, see Chapter 10, *Designing Screens With FIDEL*. You can mix FREEFORM statements with PROMPT statements in one request, and one FIXFORM statement with CRTFORM statements.
- There is no limit to the number of cases you can use in a MODIFY request.
- If a request repeatedly executes a case that has a CRTFORM, the case can produce up to 75 TYPE messages. If it produces more, FOCUS aborts the request.
- If you use fields with D. and T. prefixes in TYPE statements and CRTFORMs, a MATCH or NEXT statement must precede the fields, either in the same case or in a previously executed case (but not before the TOP case).

Executing a Case at the Beginning of a Request Only: The START Case

You can have your request begin execution with an initial case that is never executed afterwards. This case is called the START case and begins with the label:

```
CASE AT START
```

You cannot branch from other cases to the START case, but you can branch from the START case to other cases. If you do not branch to another case, the START case passes control to the TOP case. Note that the START case comes after the TOP case in the text of the request.

The following request counts how many employee salaries it updates. However, it starts counting from three:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
    SALCOUNT/I4 = SALCOUNT + 1;
  ON MATCH UPDATE CURR_SAL
TYPE AT END
  "<SALCOUNT SALARIES PROCESSED"

CASE AT START
COMPUTE
  SALCOUNT = 3;
ENDCASE
DATA
```

The START case initializes the SALCOUNT counter to 3. After that, the request does not need to refer to the case again.

Note that temporary fields used in the START case that appear earlier in the request must have their formats defined there.

Branching to Different Cases: The GOTO, PERFORM, and IF Statements

Three statements branch to other cases:

- The GOTO statement, which branches unconditionally to another case. After the case executes, control returns to the TOP case.
- The PERFORM statement, which branches unconditionally to another case. When the case called by the PERFORM reaches ENDCASE, control returns to the statement following the PERFORM.
- The IF statement, which branches to GOTO or PERFORM as above, depending on the value of a logical expression.

Syntax How to Branch to Another Case With GOTO

GOTO statements unconditionally branch to another case. The syntax is

```
GOTO location
```

where:

location

Is one of the following:

TOP branches to the beginning of the TOP case.

ENDCASE branches to the end of the case. If the case was called by a **PERFORM** statement either directly or indirectly (for example, a **PERFORM** statement called a case that branched to this case), then control returns to the statement after the most recently executed **PERFORM** statement. Otherwise, the request branches back to the TOP case.

casename branches to the beginning of the specified case.

variable branches to the beginning of the case whose name is the value of the temporary field *variable*. The temporary field must have a format of A12.

EXIT terminates the request. This is useful when you want to halt execution before the last transaction in a data source or the transaction specified by the **STOP** command. Note that the statement **GOTO EXIT** is legal even in **MODIFY** requests without cases.

If a case does not have a **GOTO** statement, it branches to the TOP case upon completion unless a **PERFORM** or **IF** statement branches somewhere else.

Syntax **How to Use a PERFORM Statement**

The PERFORM statement causes the request to branch to another case, executes that case, then returns control to the statement after the most recently executed PERFORM statement. The syntax is

```
PERFORM location
```

where:

location

Is one of the following:

TOP branches to the beginning of the TOP case. All return points are cleared and the procedure continues as if no PERFORM statement had executed.

ENDCASE branches to the end of the case. If the case was called by another PERFORM statement, either directly or indirectly (for example, a PERFORM statement called a case that branched to this case), then control returns to the statement after the most recently executed PERFORM statement. Otherwise, the request branches back to the TOP case.

casename branches to the beginning of a specified case.

variable branches to the beginning of the case whose name is the value of the temporary field *variable*. The temporary field must have a format of A12.

EXIT terminates the request.

A PERFORM statement can branch to a case containing a GOTO or IF statement that branches to a second case. The second case can branch to a third case, and so on until the request encounters an ENDCASE statement at the end of a case. Control then returns to the statement after the most recently executed PERFORM statement.

A PERFORM statement can branch to a case containing a PERFORM statement that leads to other cases. When the request encounters an ENDCASE statement at the end of a case, control returns to the statement after the most recently executed PERFORM statement. Control eventually returns to the original PERFORM.

If a case branches to the TOP case, control does not return to the last PERFORM. Rather, the request begins a new cycle starting from the TOP case. All PERFORM return points are cleared.

Example Using the PERFORM Statement

This sample request updates employee salaries. If a user enters a salary greater than \$50,000, the request checks the employee ID against a list of IDs in the sequential data source EMPLIST. If the employee is listed, the request updates the salary; otherwise, it asks the user to re-enter the information. The request is:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
PERFORM EMPCHECK
PERFORM UPSAL
TYPE
    "SALARY OF EMPLOYEE <EMP_ID UPDATED"

CASE EMPCHECK
IF CURR_SAL LE 50000 GOTO ENDCASE;
COMPUTE
    RAISE_OK/A3 = DECODE EMP_ID (EMPLIST ELSE 'NO');
IF RAISE_OK IS 'NO' THEN PERFORM TOP;
ENDCASE

CASE UPSAL
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE CURR_SAL
ENDCASE
DATA

```

Supposing the data source EMPLIST contained the following data:

```

071382660 YES
451123478 YES

```

A sample execution might go as follows:

1. The request prompts you for an employee ID and a salary. You enter ID 818692173 and a salary of \$35,000.
2. The PERFORM EMPCHECK statement branches to the EMPCHECK case.
3. Since the salary is less than \$50,000, the PERFORM ENDCASE phrase returns control to the statement after the PERFORM EMPCHECK statement (PERFORM UPSAL).
4. The PERFORM UPSAL statement branches to the UPSAL case.
5. The case updates the salary and passes control to the TYPE statement (the statement after the most recently executed PERFORM statement).
6. The TYPE statement displays the message:

```
SALARY FOR EMPLOYEE 8188692173 UPDATED
```
7. Control goes to the beginning of the TOP case.

8. The TOP case prompts you for an employee ID and a salary.
9. You enter an ID Of 119329144 and a salary of \$65,000.
10. The PERFORM EMPCHECK statement branches to the EMPCHECK case. Since employee 119329144 is not listed in the EMPLIST data source, the IF...GOTO TOP phrase branches to the TOP case.
11. The TOP case prompts you for an employee ID and a salary. You enter an ID of 071382660 and a salary of \$65,000.
12. The PERFORM EMPCHECK statement branches to the EMPCHECK case. Since employee 071382660 is listed in the EMPLIST data source, control returns to the statement after the most recently executed PERFORM statement (PERFORM UPSAL).
13. The PERFORM UPSAL statement branches to the UPSAL case, which updated the salary. Control then passes to the TYPE statement (the statement after the most recently executed PERFORM statement).
14. The TYPE statement displays a message:

```
SALARY FOR EMPLOYEE 071382660 UPDATED
```
15. Control goes to the beginning of the TOP case.

Reference Rules for PERFORM Statements

- PERFORM statements can be nested; that is, one PERFORM statement can call a case containing a second PERFORM statement. PERFORM statements can be nested to any depth, limited only by available memory. If memory runs out, FOCUS displays the message:

```
(FOC187) PERFORMS NESTED TOO DEEPLY
```

- REPEAT statements can contain PERFORM statements. When control returns to the statement after the most recently executed PERFORM statement, the REPEAT statement resumes execution. For example:

```
REPEAT 5 TIMES
  PERFORM ANALYSIS
  COMPUTE AMOUNT/D8.2 = RECEIPTS + AWARDS;
ENDREPEAT
```

Each pass of this REPEAT statement executes the ANALYSIS case, then computes the value of the AMOUNT field.

- When a PERFORM statement branches to a case, you can return control to the PERFORM before the end of the case by including the GOTO ENDCASE or PERFORM ENDCASE statement in the case.

Syntax **How to Branch to Another Case With IF**

The IF statement branches to another case depending on how an expression is evaluated. The syntax is

```
IF expr [THEN] {GOTO|PERFORM} location1 [ELSE {GOTO|PERFORM} location2]
```

where:

expr

Is any logical expression legal in a DEFINE or COMPUTE IF statement (see the *Creating Reports* manual). For example:

```
IF CURR_SAL GT 50000
IF SALARY/12 LT GROSS
IF LAST_NAME CONTAINS 'BLACK'
IF (CURR_SAL GT SALARY) OR
   (CURR_JOB CONTAINS 'B')
```

Note that literals must be enclosed in single quotation marks. Parentheses are necessary if the expression is compound.

IF expressions cannot compare data source fields unless they are used in or following MATCH or NEXT statements (see *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 9-137).

location1, location2

The options are:

TOP branches to the TOP case.

ENDCASE branches to the end of the case (the request then branches to the TOP case or to the statement after the most recently executed PERFORM statement).

case1 branches to the case named *case1*.

var branches to the case whose name is contained in the temporary field *var*.

EXIT terminates the request.

The word THEN is optional and is there to enhance readability.

An IF statement can extend over several lines, but must end with a semicolon (;).

Like IF statements in TABLE requests and Dialogue Manager control statements, Case Logic IF statements can be nested. You can nest IF statements so that if the outer IF expression is true, the inner IF is executed. Place the inner IF phrase within parentheses following the THEN phrase.

Example IF Statement

```
IF expression1
THEN (IF expression2
THEN (IF expression3 GOTO case4 ELSE GOTO case3)
ELSE GOTO case2)
ELSE GOTO case1;
```

You can also nest IF statements so that if the outer IF expression is false, the inner IF is executed. You place the inner IF statement after the ELSE phrase. The inner IF does not need parentheses:

```
IF expression1 THEN GOTO case1
ELSE IF expression2 THEN GOTO case2
ELSE IF expression3 THEN GOTO case3
ELSE...;
```

The following request offers the user a choice between deleting a segment instance and including a new one:

```
MODIFY FILE EMPLOYEE
COMPUTE CHOICE/A6=;
TYPE
  "ENTER 'UPDATE' TO UPDATE A SALARY"
  "ENTER 'DELETE' TO DELETE AN EMPLOYEE"
PROMPT CHOICE

  IF CHOICE IS 'UPDATE' THEN GOTO UPDSEG
ELSE IF CHOICE IS 'DELETE' THEN GOTO DELSEG
ELSE GOTO TOP;

CASE UPDSEG
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE

CASE DELSEG
PROMPT EMP_ID
MATCH EMP_ID
  ON MATCH DELETE
  ON NOMATCH REJECT
ENDCASE
DATA
```

This request has three cases:

- The TOP case defines a variable called CHOICE, which will contain your response to its menu:

If you enter UPDATE, the request branches to the UPDSEG case.

If you enter DELETE, the request branches to the DELSEG case.

If you enter neither, it reprompts you for another response by branching back to the beginning of the case.

- The UPDSEG case prompts you for the employee ID and new salary, and updates the employee's salary.
- The DELSEG case prompts you for the employee ID, and deletes that ID from the data source.

Rules Governing Branching

The following rules govern the sequence of case execution and branching:

- The request first executes the START case, if there is one. It then executes the TOP case, unless the START case branches to another case.
- If a case does not execute a GOTO statement, a PERFORM statement, or an IF statement to branch to another case, it branches to the TOP case by default. This is true of both the START and TOP cases. However, if the case was called by a PERFORM statement either directly or indirectly (for example, a PERFORM statement called a case that branched to a case that branched to this case), then control returns to the statement after the most recently executed PERFORM statement.
- A case can branch to itself.
- Branching to the TOP case, whether by a GOTO TOP statement, PERFORM TOP statement or by default, deactivates all data fields (field activation and deactivation are described in *Active and Inactive Fields* on page 9-199) and increments the transaction counter by one.
- When you branch to a case, you always branch to the beginning of the case. You can never branch into the middle of a case.

- If one case contains a MATCH or NEXT statement that selects a particular segment instance, it can branch to another case that modifies the child segment chain belonging to the same instance. The second case need not reselect the parent instance, but it must contain at least one MATCH statement. For example, the segment EMPINFO (key field EMP_ID) has the child segment SALINFO (key field PAY_DATE). You can include a new SALINFO segment with this request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH GOTO NEWPAY

CASE NEWPAY
MATCH PAY_DATE
    ON NOMATCH INCLUDE
    ON MATCH REJECT
ENDCASE
DATA
```

The second case, NEWPAY, modifies the segment chain descended from the segment instance selected in the TOP case.

GOTO, PERFORM, and IF Phrases in MATCH Statements

You can use GOTO, PERFORM, and IF statements in MATCH and NEXT statements, where they form part of ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrases. IF phrases in MATCH and NEXT statements can use data source fields in expressions. To do this, affix the D. prefix to the field name. For example, the phrase

```
ON MATCH IF CURR_SAL LT D.CURR_SAL . . .
```

tests whether the incoming value of CURR_SAL is less than the data source value of CURR_SAL. The data source value must either be in the segment instance that the MATCH or NEXT statement is processing or in a parent instance along the segment path (the parent, the parent's parent, and so on, up to the root segment).

For example, this request does not accept a new salary for an employee if it is less than the employee's present salary:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH IF CURR_SAL LT D.CURR_SAL GOTO ERROR;
  ON MATCH UPDATE CURR_SAL

CASE ERROR
TYPE
  "YOU ENTERED A NEW SALARY"
  "LESS THAN THE EMPLOYEE'S PRESENT SALARY"
  "PLEASE REENTER DATA"
ENDCASE
DATA
```

This request consists of two cases:

- The TOP case prompts you for an employee ID and new salary. If the employee ID is in the data source, the case tests whether the new salary is less than the present one. If the new salary is lower, it branches to the ERROR case. Otherwise, it updates the salary and branches back to the TOP case.
- The ERROR case warns you that the salary you entered is unacceptable and branches back to the TOP case.

If the MATCH statement specifies fields in multiple segments (the technique of matching across segments, described in *Modifying Segments in FOCUS Structures* on page 9-71), the GOTO, PERFORM and IF phrases in the statement are only executed when the MATCH statement modifies the last segment. For example, this request adds instances to the EMPINFO, SALINFO, and DEDUCT segments:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE DED_CODE
GOTO ADD

CASE ADD
MATCH EMP_ID PAY_DATE DED_CODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
ENDCASE

CASE MESSAGE
TYPE
  "NEW INSTANCE ADDED"
ENDCASE
DATA

```

The ADD case branches to the MESSAGE case only when it includes a new instance in the segment containing the DED_CODE field. If you want the case to branch to the MESSAGE case when it includes a new instance in any of the segments, then write the case with a separate MATCH statement for each segment it searches:

```

CASE ADD
MATCH EMP_ID
  ON MATCH CONTINUE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
MATCH PAY_DATE
  ON MATCH CONTINUE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
MATCH DED_CODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
ENDCASE

```

Example Using Case Logic and Validation Tests

You can also branch to other cases when an incoming field value fails a validation test. Do this by including GOTO, PERFORM, and IF statements as part of the ON INVALID phrase. For example, this request processes transactions with salaries higher than \$50,000 in a separate case:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL

CASE NEWSAL
PROMPT CURR_SAL
VALIDATE
    SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
    ON INVALID GOTO HIGHSAL
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH REJECT
ENDCASE

CASE HIGHSAL
TYPE
    "SALARY ABOVE $50,000 NOT ALLOWED"
    "RETYPE SALARY BELOW"
GOTO NEWSAL
ENDCASE
DATA
```

Case Logic Applications

This section discusses some examples of applications for Case Logic that extend the capabilities of MODIFY requests. The applications are:

- Looping through segment chains using the NEXT statement.
- Modifying multiple unique segments.
- Using Case Logic to offer user choices.
- Using Case Logic to process transaction data sources.
- Using Case Logic to process transactions based on the values of their fields.
- Using Case Logic to process transactions with bad values.

Syntax How to Loop Through a Segment Chain With the NEXT Statement

The NEXT statement, discussed in *Selecting the Instance After the Current Position: The NEXT Statement* on page 9-88, modifies or displays the next segment instance after the current position in the data source. Using Case Logic, you can use NEXT statements to process entire segment chains.

For an entire segment chain to be displayed, the request must branch back to the beginning of the NEXT statement. Put the NEXT statement in a separate case, as shown below:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE
    "WAGES PAID TO EMPLOYEE #<EMP_ID"
  ON MATCH GOTO SALHIST

CASE SALHIST
NEXT DAT_INC
  ON NEXT TYPE "<D.DAT_INC <D.SALARY"
  ON NEXT GOTO SALHIST
  ON NONEXT GOTO TOP
ENDCASE
DATA

```

This request consists of two cases:

- The TOP case prompts you for an employee ID and branches to the SALHIST case.
- The SALHIST case contains one NEXT statement that displays the next instance of the employee's salary chain. The case then branches back to the its beginning to display the next instance. When it reaches the end of the chain, it branches back to the TOP case.

To return to the beginning of a segment chain, use the REPOSITION statement. The syntax is

```
REPOSITION field
```

where *field* is any field of the segment. The REPOSITION statement allows you to return to the beginning of the segment chain you are now modifying, or to the beginning of the chain of any of the parent instances along the segment path (that is, the parent instance, the parent's parent, and so on to the root segment). You can then search the segment chain from the beginning.

The following request allows you to allocate a new monthly pay for a selected employee for each pay date. The request accumulates each pay in a total. If this total pay exceeds the employee's yearly salary, the request returns to the first pay date to permit you to enter new values for the entire chain:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO PAYLOOP
CASE PAYLOOP
NEXT PAY_DATE
  ON NONEXT GOTO TOP
  ON NEXT TYPE
    "EMPLOYEE ID: <EMP_ID"
    "PAY DATE: <D.PAY_DATE MONTHLY PAY: <D.GROSS"
  ON NEXT PROMPT GROSS. ENTER MONTHLY PAY: .
  ON NEXT COMPUTE
    TOTAL_PAY/D10.2 = TOTAL_PAY + GROSS;
  ON NEXT IF TOTAL_PAY GT D.CURR_SAL GOTO ERROR;
  ON NEXT UPDATE GROSS
  ON NEXT GOTO PAYLOOP
ENDCASE

CASE ERROR
TYPE
  "TOTAL MONTHLY PAY EXCEEDS YEARLY SALARY"
  "REENTER PROPOSED PAY STARTING FROM"
  "THE FIRST PAY DATE"
REPOSITION PAY_DATE
COMPUTE TOTAL_PAY = 0;
GOTO PAYLOOP
ENDCASE
DATA
```

Note that the ERROR case in the example warns you that the sum of the figures you entered exceeds the employee's yearly salary. It then repositions the current position of the PAY_DATE field at the beginning of the segment chain and branches back to the PAYLOOP case, allowing you to reenter pay figures for the entire chain.

When you use INCLUDE, UPDATE, and DELETE actions in looping NEXT statements, note the following:

- Use the ON NEXT INCLUDE and ON NONEXT INCLUDE phrases only to add instances to segments of type S0 or blank. If you use these phrases to modify other segments, you will duplicate what is already there. The difference between the two phrases is:
 ON NEXT INCLUDE adds a new segment instance after the current position.
 ON NONEXT INCLUDE adds a new instance at the end of the segment chain.
- Use the ON NEXT UPDATE phrase without restriction. The phrase updates the segment instance at the current position. If you are looping with the NEXT statement, the phrase updates the entire chain.
- Use the ON NEXT DELETE phrase to delete entire segment chains. This phrase deletes the segment instance at the current position. If you are looping with the NEXT statement, the phrase deletes the entire chain, but only if you start at the beginning of a chain. Otherwise, the phrase deletes every second instance.

Note that the phrases ON NONEXT UPDATE and ON NONEXT DELETE are illegal and will generate error messages.

Example Modifying Multiple Unique Segments

Modifying unique segments is described in *Modifying Segments in FOCUS Structures* on page 9-71. This section describes how to modify several unique segments descended from one parent using the CONTINUE TO method.

To modify multiple unique segments, prepare separate cases containing a MATCH or NEXT statement for each segment you are modifying. The sample request below illustrates this. The request loads data into the SUBSCRIBE data source, which records magazine subscribers, their mailing addresses, and expiration dates. The Master File is:

```
FILE=SUBSCRIB ,SUFFIX=FOC,$
SEGMENT=SUBSEG , $
  FIELD=SUBSCRIBER ,ALIAS=NAME ,FORMAT=A35 , $
SEGMENT=ADDRSEG, SEGTYPE=U, PARENT=SUBSEG , $
  FIELD=ADDRESS ,ALIAS=ADDR ,FORMAT=A40 , $
SEGMENT=EXPRSEG, SEGTYPE=U, PARENT=SUBSEG , $
  FIELD=EXPR_DATE ,ALIAS=EXDATE ,FORMAT=I6DMYT , $
```

The following MODIFY request loads the data:

```
MODIFY FILE SUBSCRIB
PROMPT SUBSCRIBER
MATCH SUBSCRIBER
    ON NOMATCH INCLUDE
    ON MATCH CONTINUE
GOTO NEWADDR

CASE NEWADDR
PROMPT ADDRESS
MATCH SUBSCRIBER
    ON NOMATCH REJECT
    ON MATCH CONTINUE TO ADDRESS
    ON MATCH REJECT
    ON MATCH GOTO NEWDATE
    ON NOMATCH INCLUDE
    ON NOMATCH GOTO NEWDATE
ENDCASE

CASE NEWDATE
PROMPT EXPR_DATE
MATCH SUBSCRIBER
    ON NOMATCH REJECT
    ON MATCH CONTINUE TO EXPR_DATE
    ON MATCH REJECT
    ON NOMATCH INCLUDE
ENDCASE
DATA
```

Note the last two cases in the request:

- The NEWADDR case loads subscriber addresses into the unique segment ADDRSEG. The case examines the ADDRSEG segment. Does the subscriber have a mailing address listed? If not, the request includes the new address. In either event, the request continues to the NEWDATE case.
- The NEWDATE case loads expiration dates into the sibling unique segment EXPRSEG. It examines the EXPRSEG segment with the EXPR_DATE field. Does the subscriber have a magazine expiration date? If not, the request includes the new expiration date. If the subscriber has an expiration date, the request checks to determine whether it gave the subscriber a new address.
If the request gave the subscriber a new address, the request does not reject the transaction.
If the request did not give the subscriber a new address, the request rejects the transaction.

If you were to include the MATCH statements in one case, the request would reject a transaction if the subscriber already had either an address or an expiration date. Since you want the transaction rejected only if the subscriber already has both, separate the MATCH statements into separate cases.

Procedure How to Use Case Logic to Offer User Selections

You can use Case Logic to offer users a selection of options. The request below offers a choice between updating employee salaries, monthly pay, or addresses:

```

MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH GOTO MENU

CASE MENU
TYPE
"TO UPDATE THE EMPLOYEE'S SALARY, TYPE 'SALARY' "
"TO UPDATE THE EMPLOYEE'S MONTHLY PAY, TYPE 'PAY' "
"TO UPDATE THE EMPLOYEE'S ADDRESS, TYPE 'ADDRESS' "
COMPUTE CHOICE/A7=;
PROMPT CHOICE
    IF CHOICE IS 'SALARY' THEN GOTO SALARY
    ELSE IF CHOICE IS 'PAY' THEN GOTO PAY
    ELSE IF CHOICE IS 'ADDRESS' THEN GOTO ADDRESS;
TYPE "ILLEGAL CHOICE, PLEASE TYPE ENTRY AGAIN"
GOTO MENU
ENDCASE

CASE SALARY
PROMPT CURR_SAL
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE CURR_SAL
ENDCASE

CASE PAY
PROMPT PAY_DATE GROSS
MATCH PAY_DATE
    ON NOMATCH REJECT
    ON MATCH UPDATE GROSS
ENDCASE

CASE ADDRESS
PROMPT TYPE ADDRESS_LN1 ADDRESS_LN2
MATCH TYPE
    ON NOMATCH REJECT
    ON MATCH UPDATE ADDRESS_LN1 ADDRESS_LN2
ENDCASE
DATA

```

Procedure How to Use Case Logic to Process Transaction Data Sources

You can use Case Logic to process records in a transaction data source in different ways. For example, each transaction record contains a field that defines what type of record it is. The MODIFY request can use these record types to branch to the appropriate case and process the transaction.

The following request processes two record types: type A updates employee department assignments and job codes; type B updates salaries and classroom hours. The record type field (called RTYPE) is the last field in each record. It contains either the letter A or B, depending on the record type.

```

MODIFY FILE EMPLOYEE
COMPUTE RTYPE/A1=;
FIXFORM X26 RTYPE/1
    IF RTYPE IS 'A' THEN GOTO TYPE_A
    ELSE IF RTYPE IS 'B' THEN GOTO TYPE_B;
TYPE "BAD RECTYPE VALUE"
GOTO TOP

CASE TYPE_A
FIXFORM X-27 EMP_ID/9 X1 DEPARTMENT/10
FIXFORM X1 CURR_JOBCODE/3 X3
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE DEPARTMENT CURR_JOBCODE
ENDCASE

CASE TYPE_B
FIXFORM X-27 EMP_ID/9 X1 CURR_SAL/8 X1 ED_HRS/6 X2
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH UPDATE CURR_SAL ED_HRS
ENDCASE
DATA ON FIXTYPE
END

```

Notice the three FIXFORM statements: one in each of the cases. Only the statement in the TOP case reads a record from disk or tape. The other two statements redefine the record for the case.

Also note that each of these two statements begins with X-27, which allows the case to redefine the 27-byte record from the beginning. Always place the notation X-*n* at the beginning of the FIXFORM statement that is redefining the record, not at the end of the previous FIXFORM statement.

A FIXFORM statement reads a new record from disk or tape if one of these conditions are met:

- The statement is the first FIXFORM statement in the request.
- The statement defines records to be longer than they were defined before. For instance, if one FIXFORM statement defines a record of 80 bytes, and the next FIXFORM statement defines a record from the same data source as being 90 bytes, the second FIXFORM statement reads a new record.
- The statement reads records from a different data source than the one read previously. This is possible if the statement has the form

```
FIXFORM ON ddname
```

where *ddname* is the ddname of the second transaction data source. If the next FIXFORM statement does not have the ON ddname option, it too reads another record.

Procedure How to Use Case Logic to Process Transactions Based on the Values of Their Fields

You can use Case Logic to process transactions depending on their field values. The following request updates employee salaries. If the user enters a salary higher than \$50,000, the request checks the employee ID against a list of employees authorized for large salaries:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL

CASE NEWSAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT CURR_SAL
  ON MATCH IF CURR_SAL GT 50000 THEN GOTO HIGHSAL;
  ON MATCH UPDATE CURR_SAL
ENDCASE

CASE HIGHSAL
COMPUTE
  SALTEST = DECODE EMP_ID (HIGHPAY);
IF SALTEST NE 1 THEN GOTO WRONGSAL;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
ENDCASE

CASE WRONGSAL
TYPE
  "EMPLOYEE NOT AUTHORIZED FOR SALARY INCREASE"
  "PLEASE REENTER THE DATA"
ENDCASE
DATA
```

Procedure How to Use Case Logic to Process Transactions With Bad Values

You can use Case Logic to process transactions with values that would otherwise cause the transactions to be rejected. You do this by combining GOTO and IF phrases with:

- The ON MATCH phrase, if you are adding new segment instances.
- The ON NOMATCH phrase, if you are updating or deleting instances.
- The ON INVALID phrase, if you are validating incoming data fields.

This request updates employee salaries. If it cannot find an employee record, it queries the user whether to include the transaction as a new employee record:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
    ON MATCH UPDATE CURR_SAL
    ON NOMATCH GOTO QUERY

CASE QUERY
COMPUTE CHOICE/A1=;
TYPE
    "EMPLOYEE ID NOT FOUND IN THE DATABASE"
    "INCLUDE THE TRANSACTION ANYWAY (Y/N)?"
PROMPT CHOICE
    IF CHOICE IS 'Y' THEN GOTO INCLUDE
    ELSE IF CHOICE IS 'N' THEN GOTO REJECT;
TYPE "PLEASE TYPE EITHER Y OR N"
GOTO QUERY
ENDCASE

CASE INCLUDE
MATCH EMP_ID
    ON MATCH REJECT
    ON NOMATCH INCLUDE
ENDCASE

CASE REJECT
MATCH EMP_ID
    ON MATCH REJECT
    ON NOMATCH REJECT
ENDCASE
DATA
```

Tracing Case Logic: The TRACE Facility

The TRACE facility displays the name of each case that is entered during the execution of a MODIFY request. This is a useful tool for debugging large Case Logic requests.

You can allocate the output to a file or to your terminal. Then, add the word TRACE to the end of the MODIFY command line

```
MODIFY FILE filename TRACE
```

where *filename* is the name of the FOCUS data source you are modifying.

When the TRACE facility is on, it lists in the HLIPRINT file the name of the case about to run

```
TRACE ==> AT CASE case
```

where *case* is the name of the case. Note that if you are using FIDEL and displaying the TRACE output on the terminal, the following happens. When you enter a CRTFORM screen, the screen clears and displays the name of the next case. Clear the screen, and the next CRTFORM screen appears.

The request and sample execution below illustrate the use of the TRACE facility:

```
MODIFY FILE EMPLOYEE TRACE
PROMPT EMP_ID CURR_SAL
IF CURR_SAL GT 50000 GOTO HIGHSAL
ELSE GOTO UPDATE;

CASE UPDATE
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE

CASE HIGHSAL
TYPE
  " "
  "YOU ENTERED A SALARY ABOVE $50,000"
  " "
PROMPT CURR_SAL.PLEASE REENTER THE SALARY.
IF CURR_SAL GT 50000 GOTO HIGHSAL
ELSE GOTO UPDATE;
ENDCASE
DATA
```

The following is a sample execution of the previous request:

```
> EMPLOYEE      ON 10/04/98 AT 14.02.33
**** START OF TRACE ****
TRACE ==> AT CASE TOP
DATA FOR TRANSACTION  1

EMP_ID  = > 112847612
CURR_SAL  = > 67000
TRACE ==> AT CASE HIGHSAL

YOU ENTERED A SALARY ABOVE $50,000

PLEASE REENTER THE SALARY > 27000
TRACE ==> AT CASE UPDATE
TRACE ==> AT CASE TOP
DATA FOR TRANSACTION  2

EMP_ID = 0
```

Multiple Record Processing

Multiple record processing enables you to process multiple segment instances at one time. One important application is the use of multiple record processing with the FIDEL facility to enable the terminal operator to add, update, or delete several segment instances on one screen. This section discusses multiple record processing based on this application. However, you can apply the principles stated here to other applications as well.

Usually, a MODIFY request using FIDEL prompts you for a key field value, then uses the value to retrieve one segment instance. After you modify the instance, you enter the key field value to retrieve the next instance. This way, you modify segment instances one at a time.

Multiple record processing causes the request to retrieve multiple segment instances before FIDEL displays instance values. Each time the request retrieves an instance, it stores the instance values in a work area in memory called the Scratch Pad Area. The request continues to retrieve instances until it reaches a specified number.

After the request has retrieved the instances, FIDEL reads the instance values from the Scratch Pad Area and displays them all on one screen. The user can update these values and transmit the updated values back to the data source with one press of the Enter key.

Note: Text fields cannot be put into the Scratch Pad (HOLD).

You may also design a request that adds several instances at one time, or a request that both updates existing instances and adds new ones all on the same screen.

The REPEAT Method on page 9-159 describes multiple record processing using the REPEAT statement. This method requires only that you know the fields you want to process. However, it only enables you to process instances from one segment at a time.

Manual Methods on page 9-171 discusses manual methods that require you to know how instances are stored in the Scratch Pad Area. These methods are more powerful and enable you to process multiple segments at one time.

The REPEAT Method

One REPEAT statement collects segment instances and loads them into the Scratch Pad Area; another REPEAT statement retrieves the instances from the Area and uses them to modify the data source. This method does not require you to know how the instances are stored in the Area; however, you must process the instances sequentially, and you can process only one segment at one time.

Multiple record processing has four phases. They are:

1. **Selection.** The request selects the parent instance of the instances to be processed.
2. **Collection.** The request retrieves multiple segment instances and stores their data values in the Scratch Pad Area.
3. **Display.** The FIDEL facility displays the data on one screen for editing.
4. **Modification.** The request uses the edited data values to modify the data source.

The Selection Phase: Selecting the Parent Instance

To modify multiple instances in a segment, you must first identify the parent instance. (If you are modifying the root segment, skip this phase and start with *The Collection Phase: Storing Instances in a Buffer* on page 9-160.) Do this as you would any other request.

For example, the beginning of this request identifies an employee ID in the EMPLOYEE data source, allowing you to modify the employee's child segment instances:

```
MODIFY FILE EMPLOYEE
CRTFORM LINE 2
"*****"
"* MONTHLY PAY UPDATE          *"
"*****"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO COLLECT
```

If you are using multiple record processing only to create new instances, skip the collection phase and proceed directly to the display phase. The following MATCH statement adds a new employee ID to the data source. It then branches to the case NEWADDRESS where the display phase prompts the user for all the employees' addresses:

```
MODIFY FILE EMPLOYEE
CRTFORM
"ENTER EMPLOYEE'S ID: <EMP_ID"
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO NEWADDRESS
```

The Collection Phase: Storing Instances in a Buffer

During the collection phase, the request retrieves multiple segment instances and stores their values in the Scratch Pad Area.

After identifying the parent instance, read the child instances into the Scratch Pad Area (if you are modifying the root segment, reading the instances into the Area is your first step). You do this using the REPEAT statement, which the request executes repeatedly. Each time the request executes a REPEAT statement, the phrases in the statement retrieve one segment instance and store its data values in the Area.

Syntax **How to Use a REPEAT Statement**

The syntax of the REPEAT statement is

```
REPEAT {*|count} [TIMES] [MAX limit] [NOHOLD]
.
.
phrases
.
.
ENDREPEAT
```

where:

count

Is an integer or temporary integer field determining the number of times the request executes the REPEAT. This value can be between 0 and 32,767 but should be no smaller than the number of segment instances you want to display on the FIDEL screen.

If this value is 0, the request does not execute the REPEAT (this allows you to skip a REPEAT if you are using a temporary field for this parameter). If the value is an asterisk, the REPEAT is executed 65,535 times. Once the REPEAT begins execution, the value cannot be changed.

TIMES

Is an optional word, which you can add to enhance readability.

limit

Is an integer specifying the maximum number of times the request can execute the REPEAT. Specify this parameter only if you are using a temporary field for the *count* parameter.

NOHOLD

Is an option that enables you to use REPEAT as a simple loop that executes any group of MODIFY statements repeatedly.

phrases

Are the MODIFY statements to be executed within the REPEAT statement. Each phrase must begin on a new line.

ENDREPEAT

Ends the statement. This phrase must be on a line by itself.

There are three types of REPEAT statements:

- Stacking REPEAT statements. These statements contain HOLD phrases that stack data into the Scratch Pad Area. They appear in the collection phase of multiple record processing.
- Retrieving REPEAT statements. These statements retrieve data placed in the Scratch Pad Area by the stacking REPEAT statements. They usually appear in the modification phase and in validation routines in multiple record processing.
- Simple REPEAT statements. These statements consist of any combination of MODIFY statements to be executed repeatedly. You indicate a simple repeat statement by specifying the NOHOLD option in the REPEAT phrase. Simple REPEAT statements neither stack data nor retrieve data from the Scratch Pad Area.

FOCUS determines the type of REPEAT statement in the following manner:

- If the statement specifies the NOHOLD option, it is a simple REPEAT statement.
- If the statement contains a HOLD phrase, it is a stacking REPEAT statement.
- If the statement neither specifies the NOHOLD option nor contains a HOLD phrase, it is a retrieving REPEAT statement.

The REPEAT statement can stand by itself, or it can be part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. For example:

```
REPEAT 12 TIMES
```

```
ON MATCH REPEAT 6
```

```
ON NEXT REPEAT BUFCOUNT MAX 10
```

Note: You cannot nest REPEAT statements; one statement must end before another can begin.

Two GOTO phrases especially apply to REPEAT statements. They are:

- GOTO ENDREPEAT. This phrase branches processing to the end of the REPEAT statement, increments the counter by 1, and executes the request REPEAT again.
- GOTO EXITREPEAT. This phrase branches processing to the first executable statement following the REPEAT loop.

This REPEAT saves the first five pay dates and monthly pay amounts in the EMPLOYEE data source in the Scratch Pad Area:

```
CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
    ON NEXT HOLD PAY_DATE GROSS
    ON NONEXT GOTO EXITREPEAT
ENDREPEAT
GOTO DISPLAY
ENDCASE
```

Note the ON NONEXT GOTO EXITREPEAT phrase. This specifies that if there are less than five employee IDs in the segment chain, the request will branch to the next statement after the REPEAT. If the ON NONEXT phrase was not included, the request would automatically branch back to the beginning of the request.

Syntax

How to Store Instances With the HOLD Phrase

The REPEAT statement retrieves instances using MATCH and NEXT statements. Each time the REPEAT retrieves an instance, you may store the instance values in the Scratch Pad Area. Do this with the phrase

```
HOLD [SEG.]field-1 field-2 ... field-n
```

where *field-1* through *field-n* are the data fields whose values you want to save in the Scratch Pad Area. The specified fields can be data source fields or temporary fields. The data source fields must exist either in the instance or in a parent instance along the segment path (the parent of the instance, the parent's parent, and so on to the root segment). For example, the phrase

```
HOLD EMP_ID FIRST_NAME LAST_NAME CURR_SAL
```

stores the employee IDs, first and last names, and salaries of each retrieved instance in the Scratch Pad Area.

If you want to save the values of all the data fields in the instance, specify just one field with the SEG. prefix affixed to the front of the field name.

HOLD stores the fields whether they are active or inactive. To ensure that the fields placed in the Scratch Pad Area are active, use the ACTIVATE phrase described in *Active and Inactive Fields* on page 9-199.

The HOLD phrase can stand by itself, or it can be part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. If you use HOLD in ON NOMATCH and ON NONEXT phrases, you may specify only temporary fields and fields in parent instances along the segment path. If the list of fields is too long to fit on one line, repeat the word HOLD for each line you need. Some examples are:

```
HOLD EMP_ID LAST_NAME FIRST_NAME DEPARTMENT
HOLD CURR_JOBCODE ED_HRS

ON MATCH HOLD EMP_ID DEPARTMENT CURR_SAL

ON NONEXT HOLD DEPCODE
```

When a REPEAT statement containing a HOLD phrase begins execution, FOCUS clears the Scratch Pad Area of data stored from previous REPEATs.

The following is a piece of a MODIFY request that executes the collection phase:

```
CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
  ON NEXT HOLD PAY_DATE GROSS
  ON NONEXT GOTO DISPLAY
ENDREPEAT
GOTO DISPLAY
ENDCASE
```

Reference The REPEATCOUNT and HOLDCOUNT Variables

Two variables assume values during the collection phase. These are:

- The REPEATCOUNT variable. This variable contains the value of the REPEAT counter.
- The HOLDCOUNT variable. This variable contains the current number of instances stored in the Scratch Pad Area.

If you design your request with Case Logic, you can test and branch on these variables. The following IF statement branches to the TOP case if the preceding REPEAT did not retrieve any segment instances:

```
IF HOLDCOUNT EQ 0 GOTO TOP
```

Please note the following values that the REPEATCOUNT and HOLDCOUNT variables take under these circumstances:

- When a REPEAT statement begins execution, REPEATCOUNT is set to 1.
- If a REPEAT is set to execute 0 times, REPEATCOUNT is set to 0.
- If the REPEAT beginning execution contains HOLD phrases, the Scratch Pad Area is cleared and HOLDCOUNT is set to 0. If the REPEAT does not contain HOLD phrases, HOLDCOUNT is unchanged.
- At each repetition of the REPEAT, REPEATCOUNT is increased by one. After each HOLD phrase is executed, HOLDCOUNT is increased by one.
- The REPEATCOUNT variable maintains its value after the REPEAT completes execution until the next REPEAT, even if the request branched from the REPEAT with a GOTO phrase.

Note: A CRTFORM displaying records in the Scratch Pad Area can change the HOLDCOUNT value. For this reason, you may want to store the HOLDCOUNT value in a temporary field for use later in the request. For example, this COMPUTE statement saves the value of the HOLDCOUNT field in the temporary field BUFFNUMBER:

```
COMPUTE BUFFNUMBER/I5 = HOLDCOUNT;
```

The Display Phase: Displaying Instances in One CRTFORM

After the request stores the segment instance values in the Scratch Pad Area, you display the values on one screen using the FIDEL facility (see Chapter 10, *Designing Screens With FIDEL*). Since you use the same field names for all instances (multiple record processing can only modify one segment at a time), you must distinguish between instances. To do this, add subscripts to the fields using the form.

```
field(n)
```

where *n* (the subscript) is an integer greater than 0. The subscript indicates the instance that a field belongs to in the order that the instances are read from the Scratch Pad Area.

For example, this CRTFORM displays the employee IDs, departments, and salaries of five segment instances numbered 1 through 5:

```
CASE DISPLAY
IF HOLDCOUNT EQ 0 GOTO TOP;
COMPUTE
  BUFFNUMBER/I5=HOLDCOUNT;
CRTFORM LINE 9
  " MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
  " "
  " PAY DATE          AMOUNT PAID"
  " -----          -"
  "<D.PAY_DATE (1)    <T.GROSS (1) >"
  "<D.PAY_DATE (2)    <T.GROSS (2) >"
  "<D.PAY_DATE (3)    <T.GROSS (3) >"
  "<D.PAY_DATE (4)    <T.GROSS (4) >"
  "<D.PAY_DATE (5)    <T.GROSS (5) >"
GOTO UPDATE
ENDCASE
```

Note the D. prefix (display) that displays protected field values, and the T. prefix (turnaround) that displays field values to be updated. Display fields and turnaround fields are described in Chapter 10, *Designing Screens With FIDEL*. Make all turnaround fields non-conditional; that is, end the field name with a right caret.

Once you have updated the values, you can transmit all the changes at one time by pressing the Enter key. These changes update the appropriate instances in the Scratch Pad Area. The request then branches to the modification phase (the UPDATE case), where your changes are entered into the data source. The CRTFORM may then prompt you for the next parent instance or may display the next set of multiple instances for you to change.

For example, a request that updates employee's monthly pay prompts you for an employee ID. This employee has eight pay dates recorded. The screen displays the first five pay dates. Make your adjustments and press Enter. The screen displays the last three pay dates. Make your adjustments and press Enter. The request then prompts you for the next employee ID.

You may add subscripts to fields only in CRTFORMs, not in REPEATs. REPEATs that follow the CRTFORMs process the fields in the order of the instances in the Scratch Pad Area, one at a time.

Procedure **How to Position the Cursor on Specific Field Values**

You can design the request so that the cursor is automatically positioned on a particular field value on the FIDEL screen. To do this, set the CURSOR variable equal to the field name, as described in Chapter 10, *Designing Screens With FIDEL*. If the fields are subscripted, set a field called CURSORINDEX equal to the value of the subscript. For example, this COMPUTE statement places the cursor on the field CURR_SAL(3):

```
COMPUTE  
  CURSOR/A12 = 'CURR_SAL';  
  CURSORINDEX = 3;
```

These cursor-positioning variables are useful when you perform validation tests on data entered on the FIDEL screen. After the CRTFORM, write a REPEAT statement for each field you are validating. Specify as many executions for the REPEAT as the highest subscript in the CRTFORM.

In the REPEAT statement:

- Set the CURSOR variable equal to the name of the field you are validating.
- Set the CURSORINDEX variable equal to the REPEATCOUNT variable. This sets the CURSORINDEX variable to the subscript of the field being validated.
- Validate the field.
- If a field value proves invalid, branch back to the CRTFORM using Case Logic. The CURSOR and CURSORINDEX variables will position the cursor at the invalid value.

Note: Remember to assign the CURSOR variable a format of A12 and the CURSORINDEX variable a format of I5.

This is a sample case validating the CURR_SAL field:

```
CASE DISPLAY
CRTFORM
"EMPLOYEE          SALARY          DEPARTMENT"
"-----          -"
"<D.EMP_ID(1)  <T.CURR_SAL(1) >  <T.DEPARTMENT(1) >"
"<D.EMP_ID(2)  <T.CURR_SAL(2) >  <T.DEPARTMENT(2) >"
"<D.EMP_ID(3)  <T.CURR_SAL(3) >  <T.DEPARTMENT(3) >"
"<D.EMP_ID(4)  <T.CURR_SAL(4) >  <T.DEPARTMENT(4) >"
"<D.EMP_ID(5)  <T.CURR_SAL(5) >  <T.DEPARTMENT(5) >"

REPEAT 5 TIMES
  COMPUTE
    CURSOR/A12 = 'CURR_SAL';
    CURSORINDEX/15 = REPEATCOUNT;
  VALIDATE
    SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
    ON INVALID TYPE
      "THIS SALARY ENTERED WAS TOO HIGH"
      "PLEASE RE-ENTER"
    ON INVALID GOTO DISPLAY
ENDREPEAT
ENDCASE
```

The Modification Phase

After the user has entered changes on a FIDEL screen, the request uses the data to update instances in the Scratch Pad Area and to add new ones. To transfer the changes from the Area to the data source, prepare a REPEAT statement that modifies a data source instance on each pass.

This REPEAT updates the EMPLOYEE data source using data entered on the FIDEL screen shown in the previous section, *The Display Phase: Displaying Instances in One CRTFORM* on page 9-165. The REPEAT should execute as many times as there are instances in the Scratch Pad Area. This number was stored in the HOLDCOUNT variable. However, the HOLDCOUNT value can be changed by the CRTFORMs that display records in the Area. Therefore, you should store the HOLDCOUNT variable in a temporary field in the display phase before the CRTFORM. (This is shown in the example at the beginning of the section mentioned above.) This field can then set the number of times that the REPEAT statement executes.

At each pass, the REPEAT statement retrieves one instance from the Scratch Pad Area. It can then match on key fields in the instance to locate the corresponding instance in the data source (or determine that such an instance does not exist), then update the data source instance or add a new one.

In this example, the case UPDATE updates the data source instances, then branches back to the collection phase (COLLECT case). The collection phase reads the next five employee pay dates, which you can then change on the CRTFORM. This cycle continues until all the employee's pay dates have been read. You then enter the ID of the next employee. The number of instances in the Scratch Pad Area is contained in the temporary field BUFFNUMBER:

```
CASE UPDATE
REPEAT BUFFNUMBER
    MATCH PAY_DATE
        ON NOMATCH INCLUDE
        ON MATCH UPDATE GROSS
ENDREPEAT
GOTO COLLECT
ENDCASE

DATA VIA FI3270
END
```

Example **Using Multiple Record Processing (REPEAT Method)**

The sample request on the next page updates the monthly pay of employees. The CRTFORM in the display phase displays the data for the five months in which the employee was paid. After you update the monthly pay of these five months, the display phase displays the next five months. This continues until it displays all the months recorded for that employee. The request then prompts for the next employee ID.

Multiple Record Processing

The request is as follows:

```
MODIFY FILE EMPLOYEE
CRTFORM LINE 2
"*****"
"*MONTHLY PAY UPDATE*"
"*****"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO COLLECT

CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
    ON NEXT HOLD PAY_DATE GROSS
    ON NONEXT GOTO DISPLAY
ENDREPEAT
GOTO DISPLAY
ENDCASE

CASE DISPLAY
IF HOLDCOUNT EQ 0 GOTO TOP;
COMPUTE
  BUFFNUMBER/I6=HOLDCOUNT;
CRTFORM LINE 9
" MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
" "
"PAY DATE                AMOUNT PAID"
"-----                -"
"<D.PAY_DATE(1)          <T.GROSS(1)>"
"<D.PAY_DATE(2)          <T.GROSS(2)>"
"<D.PAY_DATE(3)          <T.GROSS(3)>"
"<D.PAY_DATE(4)          <T.GROSS(4)>"
"<D.PAY_DATE(5)          <T.GROSS(5)>"
GOTO UPDATE
ENDCASE

CASE UPDATE
REPEAT BUFFNUMBER
  MATCH PAY_DATE
    ON NOMATCH INCLUDE
    ON MATCH UPDATE GROSS
ENDREPEAT
GOTO COLLECT
ENDCASE

DATA VIA FI3270
END
```

Manual Methods

This section discusses manual methods of multiple record processing. These methods allow you to manipulate individual records in the Scratch Pad Area and to process instances from multiple segments at one time.

Manual methods depend on two temporary fields:

- The HOLDINDEX field. This field contains index values of records in the Scratch Pad Area. When you place a record in the Area using the HOLD statement, FOCUS assigns the record an index value equal to the value of the HOLDINDEX field. When you request a record from the Area using the GETHOLD statement, FOCUS retrieves the record having an index value equal to the value of the HOLDINDEX field.

When you place a record into the area using the HOLD phrase, set HOLDCOUNT equal to HOLDINDEX, then increment HOLDINDEX by 1.

- The SCREENINDEX field. This field determines the group of records to appear on subscripted CRTFORMs.

There are manual methods for the collection, sorting, display, and modification phases of multiple record processing. There are no manual methods for the first phase, the selection phase (discussed in *Multiple Record Processing* on page 9-158). Note, however, that if you process multiple segments that have no common parent, you must select the parent instance of each segment chain.

Initialization

Before loading instances into the Scratch Pad Area, the request may need to perform the following tasks:

- Define the following variables with a format of I5:
 - The HOLDCOUNT field. Set HOLDCOUNT equal to 0.
 - The HOLDINDEX field. Set HOLDINDEX equal to 1.
 - The SCREENINDEX field. Set SCREENINDEX equal to 0.
- Use the REPOSITION statement to insure that the current position in each segment, from which instances will be loaded into the Scratch Pad Area, is at the beginning of the segment.

The following is the beginning of a MODIFY request that uses manual methods:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5 = 0;
  HOLDINDEX/I5 = 1;
  SCREENINDEX/I5 = 0;
REPOSITION SALARY
REPOSITION PAY_DATE
GOTO SALCOLLECT
ENDCASE
```

The Collection Phase: The HOLDINDEX Field

During the collection phase, the request retrieves multiple segment instances from the data source and stores each instance as a record in the Scratch Pad Area. FOCUS assigns each record an index value equal to the current value of the HOLDINDEX field, then increments HOLDINDEX by 1. For example, if HOLDINDEX is equal to 5, then the request stores one segment instance in the Area as Record 5, the next instance as Record 6, and so on.

To store instances from multiple segments, follow this procedure:

1. Assign each segment a range of index values (for example, assign one segment values 1 through 5, another 6 through 11, and so on).
2. Write the request so that a separate case loads instances from each segment. Before each case executes, have a COMPUTE statement set HOLDINDEX to the index value of the first record for that segment.

To assign index values to a segment, you must know the largest number of instances you will be storing from that segment. In many applications, you will be storing an entire segment chain at a time. You then must know the size of the largest segment chain.

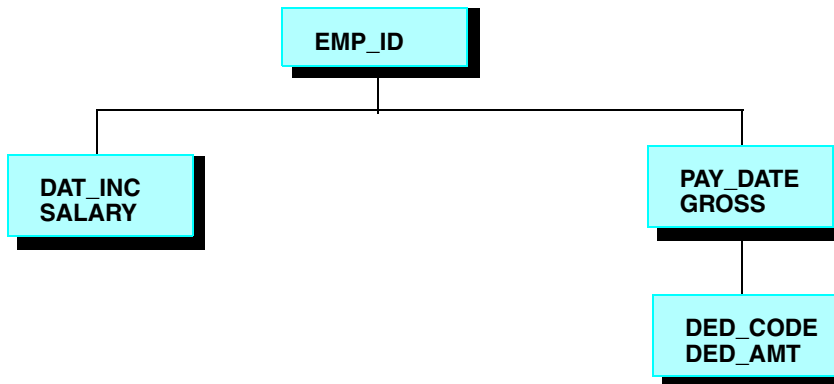
Note: Be sure that you set HOLDINDEX to a value less than or equal to the current value of the HOLDCOUNT field. A HOLDINDEX value greater than HOLDCOUNT generates an error that terminates the request.

For example, suppose you write a request to update both employees' salary history (SALARY) and monthly pay (GROSS), information contained in two different segments in the EMPLOYEE data source (see the diagram that follows).

To determine the size of the largest chains in both segments, enter this procedure:

```
TABLE FILE EMPLOYEE
COUNT SALARY AND PAY_DATE BY EMP_ID
ON TABLE HOLD
END

TABLE FILE HOLD
SUM MAX.SALARY AND MAX.PAY_DATE
END
```



The output appears as follows:

```
PAGE      1
MAX       MAX
SALARY    PAY_DATE
-----
          2      10
```

The report shows that the largest salary history chain consists of two instances and the largest monthly pay chain consists of ten instances. Therefore, you assign values 1 and 2 to the salary history segment and values 3 through 12 to the monthly pay segment. Schematically, the Scratch Pad Area will look like this:

1.	DAT_INC (1)	SALARY (1)	-	-
2.	DAT_INC (2)	SALARY (2)	-	-
3.	-	-	PAY_DATE (3)	GROSS (3)
4.	-	-	PAY_DATE (4)	GROSS (4)
5.	-	-	PAY_DATE (5)	GROSS (5)
6.	-	-	PAY_DATE (6)	GROSS (6)
7.	-	-	PAY_DATE (7)	GROSS (7)
8.	-	-	PAY_DATE (8)	GROSS (8)
9.	-	-	PAY_DATE (9)	GROSS (9)
10.	-	-	PAY_DATE (10)	GROSS (10)
11.	-	-	PAY_DATE (11)	GROSS (11)
12.	-	-	PAY_DATE (12)	GROSS (12)

To fix the index values in the request, set HOLDINDEX to the first index value assigned to a segment before loading instances from that segment. In the example above, set HOLDINDEX to 1 before loading the salary history instances, and set HOLDINDEX to 3 before loading the monthly pay instances. This reserves the proper index values for each segment.

Prepare separate cases to load instances from each segment. During the modification phase, discussed on the next page, you may plan to retrieve all records from the same segment at one time. If so, store the index value of the last instance loaded into the Scratch Pad Area from that segment (this is the HOLDINDEX value after the last instance is loaded minus one) in a field. This field will help retrieve the records in the modification phase.

For example, you are loading monthly pay instances into the Scratch Pad Area. The last monthly pay instance loaded into the Area is assigned index value 8. You then store 8 in the field LASTPAY.

This example is a request fragment that updates employees' salary histories and monthly pay:

```

CASE SALCOLLECT
NEXT SALARY
  ON NEXT HOLD DAT_INC SALARY
  ON NEXT GOTO SALCOLLECT
  ON NONEXT COMPUTE
    LASTSAL/I5 = HOLDINDEX-1;
    HOLDINDEX = 3;
  ON NONEXT GOTO PAYCOLLECT
ENDCASE

CASE PAYCOLLECT
NEXT PAY_DATE
  ON NEXT HOLD PAY_DATE GROSS
  ON NEXT GOTO PAYCOLLECT
  ON NONEXT COMPUTE
    LASTPAY/I5 = HOLDINDEX-1;
  ON NONEXT GOTO DISPLAY
ENDCASE

```

The three cases are:

- **The TOP case.** This case selects an employee and sets the HOLDINDEX field to 1 to index the salary history instances.
- **The SALCOLLECT case.** This case loads the salary history instances into the Scratch Pad Area. After the instances are loaded, the case stores the index value of the last loaded salary history instance in the field LASTSAL. It then sets the HOLDINDEX field to 3 to index the monthly pay instances.
- **The PAYCOLLECT case.** This case loads the monthly pay instances into the Scratch Pad Area. After it loads the instances, it stores the index value of the last loaded monthly pay instance in the field LASTPAY. It then proceeds to the display phase.

The Display Phase: The SCREENINDEX Field

This section shows how to display a specific group of records in the Scratch Pad Area.

The REPEAT Method on page 9-159 described how to display records in the Scratch Pad Area on a CRTFORM. The CRTFORM statement specifies the field names with subscripts that refer to the records in the Area. For example:

```
CRTFORM
"MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
" "
"PAY DATE          AMOUNT PAID"
"-----          -"
"<D.PAY_DATE (1)   <T.GROSS (1) >"
"<D.PAY_DATE (2)   <T.GROSS (2) >"
"<D.PAY_DATE (3)   <T.GROSS (3) >"
"<D.PAY_DATE (4)   <T.GROSS (4) >"
"<D.PAY_DATE (5)   <T.GROSS (5) >"
```

To display a subscripted field, FOCUS adds the field subscript to the value of a field called SCREENINDEX, then uses the sum as an index value to locate a record in the Scratch Pad Area. It then displays the field value in that record. For example, if the SCREENINDEX value for the above CRTFORM is 4, FOCUS will display the PAY_DATE and GROSS values from Area records 5 through 9.

You can use this feature to scroll back and forth through the Scratch Pad Area. To scroll forward, increase the value of SCREENINDEX; to scroll backward, decrease the value of SCREENINDEX.

If you update a field value on the CRTFORM, FOCUS updates the appropriate record in the Scratch Pad Area.

Note:

- If the request does not give SCREENINDEX a value, the default value is 0.
- If the sum of the SCREENINDEX value and a field subscript is less than 0 or more than the current value of the HOLDCOUNT field, then the CRTFORM displays that field as blank.
- If you use the CURSORINDEX field to place the cursor on a field value (as described in *The REPEAT Method* on page 9-159), the CURSORINDEX value refers to the field subscript, not the index value.

This sample case displays blocks of eight records stored in the Scratch Pad Area. The first record in each block is a monthly pay instance. The remaining seven records are deductions taken from the employee's paycheck. The case is:

```

CASE DISPLAY
IF HOLDCOUNT EQ 0 THEN GOTO TOP;
COMPUTE
  PFKEY/A4 = ' ';
  EMPID/A9 = EMP_ID;
  DED_AMT/D12.2M = DED_AMT;
CRTFORM LINE 1
  "DEDUCTION RECORD SCREEN"
  " "
"  EMPLOYEE: <D.EMPID          PAY DATE: <D.PAY_DATE(1) "
" "
"1. <D.DED_CODE(2)          <T.DED_AMT(2) >"
"2. <D.DED_CODE(3)          <T.DED_AMT(3) >"
"3. <D.DED_CODE(4)          <T.DED_AMT(4) >"
"4. <D.DED_CODE(5)          <T.DED_AMT(5) >"
"5. <D.DED_CODE(6)          <T.DED_AMT(6) >"
"6. <D.DED_CODE(7)          <T.DED_AMT(7) >"
"7. <D.DED_CODE(8)          <T.DED_AMT(8) >"
" "
"PRESS PF4 TO DISPLAY THE NEXT EMPLOYEE"
"PRESS PF5 TO DISPLAY THE LAST PAY DATE"
"PRESS PF6 TO DISPLAY THE NEXT PAY DATE"
COMPUTE
  SCREENINDEX/I5 = IF PFKEY IS 'PF04' THEN 0
                  ELSE IF PFKEY IS 'PF05' THEN SCREENINDEX - 8
                  ELSE IF PFKEY IS 'PF06' THEN SCREENINDEX + 8
                  ELSE SCREENINDEX;
IF PFKEY IS 'PF04' THEN GOTO TOP ELSE GOTO DISPLAY;

```

Pressing one of the PF keys gives the variable PFKEY a value that the request tests to adjust SCREENINDEX. By adding eight to SCREENINDEX, the request displays the next block of records. By subtracting eight from SCREENINDEX, the request displays the previous block of records.

The Modification Phase: The GETHOLD Statement

During the modification phase, the request retrieves records from the Scratch Pad Area and uses them to modify the data source. It retrieves records using the GETHOLD statement. The syntax is

GETHOLD

without any parameters. The GETHOLD statement retrieves the record whose index value is the value of the HOLDINDEX field. The HOLDINDEX field is then incremented by 1. For example, if the current value of HOLDINDEX is 5, the GETHOLD statement retrieves Record 5 from the Scratch Pad Area. HOLDINDEX is then increased to 6.

After the record is retrieved, all fields in the record become available for processing: matching, adding new segment instances, updating, deleting, and computations. Note that you may need to activate these fields before processing. For example, these statements update an employee's monthly pay using Record 5 in the Scratch Pad Area. Record 5 contains two fields: PAY_DATE and GROSS:

```
COMPUTE HOLDINDEX = 5;  
GETHOLD  
ACTIVATE PAY_DATE GROSS  
MATCH PAY_DATE  
    ON NOMATCH REJECT  
    ON MATCH UPDATE GROSS
```

You may use the GETHOLD statement to process all the records in the Scratch Pad Area. If the records contain data loaded from different segments, use separate cases to process records from each segment. First, set the HOLDINDEX field to the index value of the first record from the segment. As the request retrieves each record, HOLDINDEX increases by 1. When HOLDINDEX is greater than the index value of the last record from the segment (which you stored earlier in a field), you can branch to another case.

For example, this request fragment updates employees' salary history and monthly pay. The Scratch Pad Area consists of the following records:

- The first two records contain the fields DAT_INC and SALARY to update the salary history.
- The next ten records contain the fields PAY_DATE and GROSS to update monthly pay.

The fragment is:

```

CASE SALSET
COMPUTE HOLDINDEX = 1;
GOTO SALUPDATE
ENDCASE

CASE SALUPDATE
GETHOLD
MATCH DAT_INC
    ON MATCH UPDATE SALARY
    ON MATCH     IF HOLDINDEX GT LASTSAL GOTO PAYSET
    ELSE GOTO SALUPDATE;
    ON NOMATCH REJECT
ENDCASE

CASE PAYSET
COMPUTE HOLDINDEX = 3;
GOTO PAYUPDATE
ENDCASE

CASE PAYUPDATE
GETHOLD
MATCH PAY_DATE
    ON MATCH UPDATE GROSS
    ON MATCH     IF HOLDINDEX GT LASTPAY GOTO TOP
    ELSE GOTO PAYUPDATE;
    ON NOMATCH REJECT
ENDCASE

DATA VIA FIDEL
END

```

The cases are as follows:

- The SALSET case sets HOLDINDEX to 1, the index value of the first salary history record.
- The SALUPDATE case updates the salary history using the records in the Scratch Pad Area. Each time the case retrieves a record, HOLDINDEX is incremented by 1. When HOLDINDEX is greater than the index value of the last salary history record (the value of field LASTSAL), the case branches to the PAYSET case.
- The PAYSET case sets HOLDINDEX to 3, the index value of the first monthly pay record in the Scratch Pad Area.
- The PAYUPDATE case updates monthly pay using the records in the Scratch Pad Area. When HOLDINDEX is greater than the index value of the last monthly pay record in the Area (the value of field LASTPAY), the case branches back to the top.

You can also use the GETHOLD statement to retrieve and process a single record from the Scratch Pad Area. This request fragment allows the user to delete a single monthly pay instance:

```
CASE DISPLAY
CRTFORM
COMPUTE LN/I1 = 0;
  "MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
  " "
  "PAY DATE          AMOUNT PAID"
  "-----          -"
  "1. <D.PAY_DATE (1) <T.GROSS (1) >"
  "2. <D.PAY_DATE (2) <T.GROSS (2) >"
  "3. <D.PAY_DATE (3) <T.GROSS (3) >"
  "4. <D.PAY_DATE (4) <T.GROSS (4) >"
  "5. <D.PAY_DATE (5) <T.GROSS (5) >"
  " "
  "TO DELETE AN INSTANCE, ENTER LINE NUMBER HERE: <LN"
IF (LN LT 1) OR (LN GT 5) GOTO DISPLAY ELSE GOTO DELETE;
ENDCASE

CASE DELETE
COMPUTE
  HOLDINDEX = LN;
GETHOLD
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON NOMATCH GOTO TOP
  ON MATCH DELETE
  ON MATCH GOTO TOP
ENDCASE
```

Note: Be sure that you set HOLDINDEX to a value less than or equal to the current value of the HOLDCOUNT field. A HOLDINDEX value greater than HOLDCOUNT generates an error that terminates the request.

Reference **Manual Methods: Two Examples**

This section shows two examples that illustrate manual methods in multiple record processing:

- The first example updates employees' salary history and monthly pay. This is data contained in segments on two different paths.
- The second example deletes records of employee deductions. This is data contained in segments on one path (a parent and its child).

A diagram showing the place of salary history (SALARY), monthly pay (GROSS), and pay deductions (DED_AMT) in the EMPLOYEE data source structure appears at the beginning of *The Collection Phase: The HOLDINDEX Field* on page 9-172 in this section.

Example First Example: Processing Segments on Two Different Paths

This request is an example of a procedure that processes segments lying on different paths. The example updates employees' salary history and monthly pay. The salary history segment and monthly pay segment are both children of the employee segment, and they are on two separate paths.

This request also demonstrates the use of the GETHOLD statement to retrieve segment chains from the Scratch Pad Area. Explanatory comments are embedded in the request.

```

MODIFY FILE EMPLOYEE
-* First, select the parent employee instance.

CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
-* Flush the Scratch Pad Area, then initialize fields
-* and segment chains.

REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5 = 0;
  HOLDINDEX/I5 = 1;
REPOSITION SALARY
REPOSITION PAY_DATE
GOTO SALCOLLECT
ENDCASE

CASE SALCOLLECT
-* Place the employees' salary history in the Scratch
-* Pad Area. Afterwards, store the index value of the
-* last loaded instance in the field LASTSAL. Then
-* set HOLDINDEX to 3, which is the index of the
-* first monthly pay instance.

NEXT SALARY
  ON NEXT HOLD DAT_INC SALARY
  ON NEXT GOTO SALCOLLECT
  ON NONEXT COMPUTE
    LASTSAL/I5 = HOLDINDEX-1;
  HOLDINDEX = 3;
  ON NONEXT GOTO PAYCOLLECT
ENDCASE

```

Multiple Record Processing

```
CASE PAYCOLLECT
-* Place the monthly pay instances in the Scratch Pad
-* Area. Afterwards, store the index value of the last
-* loaded instance in the field LASTPAY.

NEXT PAY_DATE
  ON NEXT HOLD PAY_DATE GROSS
  ON NEXT GOTO PAYCOLLECT
  ON NONEXT COMPUTE
    LASTPAY/I5 = HOLDINDEX-1;
  ON NONEXT GOTO DISPLAY
ENDCASE

CASE DISPLAY
-* If nothing was collected, go back to TOP.
-* Otherwise, display the two segment chains
-* side by side. Then reset HOLDINDEX to 1
-* to prepare for updating.

IF HOLDCOUNT EQ 0 GOTO TOP;
CRTFORM LINE 3
  "SALARY HISTORY AND MONTHLY PAY RECORD"
  " "
  "SALARY HISTORY                <40 MONTHLY PAY"
  "-----"                <40 -----"
  " "
  " <D.DAT_INC(1) <T.SAL(1>          <40 <D.PD(3) <T.GROSS(3) >"
  " <D.DAT_INC(2) <T.SAL(2>          <40 <D.PD(4) <T.GROSS(4) >"
  "                                     <40 <D.PD(5) <T.GROSS(5) >"
  "                                     <40 <D.PD(6) <T.GROSS(6) >"
  "                                     <40 <D.PD(7) <T.GROSS(7) >"
  "                                     <40 <D.PD(8) <T.GROSS(8) >"
  "                                     <40 <D.PD(9) <T.GROSS(9) >"
  "                                     <40 <D.PD(10) <T.GROSS(10) >"
  "                                     <40 <D.PD(11) <T.GROSS(11) >"
  "                                     <40 <D.PD(12) <T.GROSS(12) >"

COMPUTE HOLDINDEX=1;
GOTO SALUPDATE
ENDCASE

CASE SALUPDATE
-* Update the salary history instances.
-* LASTSAL contains the index value of the
-* last salary history record.

GETHOLD
MATCH DAT_INC
  ON MATCH UPDATE SALARY
  ON MATCH IF HOLDINDEX GT LASTSAL GOTO HOLDSET
  ELSE GOTO SALUPDATE;
  ON NOMATCH REJECT
ENDCASE
```



```

CASE HOLDSET
-* Set HOLDINDEX to 3 to update the first
-* monthly pay instance.

COMPUTE HOLDINDEX = 3;
GOTO PAYUPDATE
ENDCASE

CASE PAYUPDATE
-* Update the monthly pay instances. The field
-* LASTPAY contains the index value of the last
-* monthly pay record. Afterwards, go back to TOP.

GETHOLD
MATCH PAY_DATE
  ON MATCH UPDATE GROSS
  ON MATCH IF HOLDINDEX GT LASTPAY GOTO TOP
  ELSE GOTO PAYUPDATE;
  ON NOMATCH REJECT
ENDCASE

DATA VIA FIDEL
END

```

Example **Second Example: Modifying Segments on the Same Path**

This is a sample request that processes segments lying on the same path. The request deletes employee pay deductions. To do so, it displays a pay date on the top of the screen; below, it shows the deductions taken from the employee's pay check that date. The user can scroll back and forth between pay dates and may choose particular deductions to delete. The pay date is a field in the monthly pay segment; the deductions are fields in the child deduction segment, as shown in the diagram in *The Collection Phase: The HOLDINDEX Field* on page 9-172.

The request also demonstrates the use of the SCREENINDEX field to display different groups of records on subscribed CRTFORMs, and the use of the GETHOLD statement to retrieve specific records. Explanatory comments are embedded in the request.

```

MODIFY FILE EMPLOYEE

-* First, select the parent employee instance.

CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
-* Flush the Scratch Pad Area, then initialize fields
-* and segment chains.

```

Multiple Record Processing

```
REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5      = 0;
  HOLDINDEX/I5       = 1;
  SCREENINDEX/I5     = 0;
BLOCKCOUNT/I5      = 0;
REPOSITION PAY_DATE
GOTO PAYCOLLECT
ENDCASE

CASE PAYCOLLECT
- * The next two cases create blocks of eight
- * instances within the Scratch Pad Area. Each block
- * consists of a monthly pay instance followed
- * by seven descendant instances in the
- * deduction segment. The field BLOCKCOUNT counts
- * the number of blocks in the Scratch Pad Area so far.
- * The field BLOCKNUM contains the total number of
- * blocks in the Area after all instances have
- * been loaded.

NEXT PAY_DATE
  ON NEXT COMPUTE
    HOLDINDEX = 8 * BLOCKCOUNT + 1;
    BLOCKCOUNT = BLOCKCOUNT + 1;
  ON NEXT ACTIVATE PAY_DATE
  ON NEXT HOLD PAY_DATE
  ON NEXT GOTO DEDCOLLECT
  ON NONEXT COMPUTE
    BLOCKNUM/I5 = BLOCKCOUNT;
  ON NONEXT GOTO DISPLAY
ENDCASE

CASE DEDCOLLECT
NEXT DED_CODE
  ON NEXT ACTIVATE DED_CODE DED_AMT
  ON NEXT HOLD DED_CODE DED_AMT
  ON NEXT GOTO DEDCOLLECT
  ON NONEXT GOTO PAYCOLLECT
ENDCASE
```

```

CASE DISPLAY
-* If nothing was collected, go back to TOP.
-* Otherwise, initialize the PFKEY and LINENO
-* fields. The EMPID field is for display
-* purposes. Then, display the current block.
-*
-* At the bottom of the screen is a menu to offer
-* users the choice of processing the records
-* of another employee, displaying the previous
-* block or displaying the next block. the field
-* PFKEY reads the PF key that the user presses
-* (see Chapter 16). The field LINENO contains the
-* line number of the deduction instance that the
-* user wants to delete.

IF HOLDCOUNT EQ 0 THEN GOTO TOP;
COMPUTE
  PFKEY/A4      = ' ';
  LINENO/I1     = 0;
  EMPID/A9      = EMP_ID;
CRTFORM LINE 1
  "DEDUCTION RECORD DELETION SCREEN"
  " "
  "EMPLOYEE: <D.EMPID      PAY DATE: <D.PAY_DATE(1) "
  " "
  "1. <D.DED_CODE(2) <D.DED_AMT(2) "
  "2. <D.DED_CODE(3) <D.DED_AMT(3) "
  "3. <D.DED_CODE(4) <D.DED_AMT(4) "
  "4. <D.DED_CODE(5) <D.DED_AMT(5) "
  "5. <D.DED_CODE(6) <D.DED_AMT(6) "
  "6. <D.DED_CODE(7) <D.DED_AMT(7) "
  "7. <D.DED_CODE(8) <D.DED_AMT(8) "
  " "
  "PRESS PF4 TO DISPLAY THE NEXT EMPLOYEE"
  "PRESS PF5 TO DISPLAY THE LAST PAY DATE"
  "PRESS PF6 TO DISPLAY THE NEXT PAY DATE"
  " "
  "TO DELETE A RECORD, ENTER LINE NUMBER HERE ==> <LINENO"

IF PFKEY IS 'PF04' THEN GOTO TOP;
IF (LINENO GE 1) AND (LINENO LE 7) THEN PERFORM DELETE;
IF (PFKEY IS 'PF05') OR (PFKEY IS 'PF06')
  THEN PERFORM ADJUST;
GOTO DISPLAY
ENDCASE

```

Multiple Record Processing

```
CASE ADJUST
-* Adjust SCREENINDEX to display another block.
-* The BACK and FORW fields perform the arithmetic
-* but also insure that SCREENINDEX stays within
-* its proper range. BLOCKNUM is the total number
-* of blocks in the Scratch Pad Area.

COMPUTE
  BACK/I5 = IF SCREENINDEX GT 8 THEN SCREENINDEX-8 ELSE 0;
  FORW/I5 = IF SCREENINDEX LT 8*(BLOCKNUM-1)
  THEN SCREENINDEX+8 ELSE 8*(BLOCKNUM-1);
SCREENINDEX = IF PFKEY IS 'PF05' THEN BACK ELSE FORW;
ENDCASE

CASE DELETE
-* Delete the deduction instance indicated by the user.
-* The first GETHOLD statement retrieves the monthly
-* pay instance from the Scratch Pad Area. The second
-* GETHOLD statement retrieves the desired deduction
-* instance. After activating the PAY_DATE and DED_CODE
-* key fields, the case locates the deduction instance
-* in the database and deletes it. Note: The record
-* in the Scratch Pad Area is NOT deleted and will
-* continue to appear on the screen.

COMPUTE HOLDINDEX = SCREENINDEX + 1;
GETHOLD
COMPUTE HOLDINDEX = SCREENINDEX + LINENO + 1;
GETHOLD
ACTIVATE PAY_DATE DED_CODE
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DED_CODE
  ON NOMATCH TYPE "DEDUCTION RECORD NOT FOUND"
  ON NOMATCH GOTO DISPLAY
  ON MATCH DELETE
  ON MATCH TYPE "RECORD ON LINE <LINENO DELETED"
  ON MATCH GOTO DISPLAY
ENDCASE

DATA VIA FIDEL
END
```

Procedure How to Sort the Scratch Pad Area With SORTHOLD

You can sort the contents of the Scratch Pad Area using any field or combination of fields in the Scratch Pad Area; you can then display them in any convenient order. The command uses syntax similar to the sorting specifications in the TABLE command.

The MODIFY subcommand that sorts the Scratch Pad Area is

```
SORTHOLD BY [HIGHEST] field1 [BY [HIGHEST] field2...]
```

where *field1* is the primary sort field, and *field2* to *field8* are optional secondary sort fields.

Note:

- The SORTHOLD statement cannot span more than one line. The default sort order is from low-to-high, but a high-to-low sort can be specified with the keyword HIGHEST. You can sort the Scratch Pad Area by up to eight fields.
- If you sort the Scratch Pad Area before display, always sort by the data source key fields before entering a MATCH...UPDATE loop, to be sure the transactions are in sequence with the data source. Otherwise you increase execution time substantially. This procedure

```
SORTHOLD BY ITEM
```

performs this sort. It is issued after the records are displayed but before they are updated in the data source.

Consider the following Master File:

```
FILENAME=PRODUCT, SUFFIX=FOC
  SEGNAME=SEGONE, SEGTYPE=S1
    FIELD=ORDERNO, ALIAS=ONO, FORMAT=I4, $
  SEGNAME=SEGTWO, SEGTYPE=S1, PARENT=SEGONE
    FIELD=ITEM, ALIAS=ITEMNO, FORMAT=A3, $
    FIELD=PRODUCT, ALIAS=PRD, FORMAT=A12, $
    FIELD=QTY, ALIAS=QUANTITY, FORMAT=I4S, $
```

Multiple Record Processing

The following procedure will display all of the ITEM instances for a specified ORDERNO, in order of the PRODUCT name and highest QTY sequence. The command

```
SORTHOLD BY PRODUCT BY QTY
```

performs the sort.

```
MODIFY FILE PRODUCT
CRTFORM LINE 1
  "ENTER ORDER NUMBER <ORDERNO"
MATCH ORDERNO
  ON NOMATCH GOTO TOP
  ON MATCH REPEAT 12
    NEXT ITEM
      ON NEXT HOLD ITEM PRODUCT QTY
      ON NONEXT GOTO SCREEN
    ENDREPEAT
GOTO SCREEN
CASE SCREEN
  IF HOLDCOUNT EQ 0 GOTO TOP;

  SORTHOLD BY PRODUCT BY HIGHEST QTY

  CRTFORM LINE 1
    "ORDER NUMBER IS <D.ORDERNO          "
    "                                     "
    " ITEM          PRODUCT          QUANTITY "
    " ----          -"
    "<D.ITEM(1)    <D.PRODUCT(1)    <T.QTY(1) > "
    "<D.ITEM(2)    <D.PRODUCT(2)    <T.QTY(2) > "
    "
    "
    "<D.ITEM(12)  <D.PRODUCT(12)  <T.QTY(12) >"

  SORTHOLD BY ITEM
  REPEAT HOLDCOUNT
    MATCH ITEM
      ON MATCH UPDATE
  QTY
    ON NOMATCH GOTO
  ENDREPEAT
  ENDREPEAT
  GOTO TOP
ENDCASE
DATA VIA FIDEL
END
```

Advanced Facilities

The following facilities can assist you in using the MODIFY command:

- The COMBINE command, for modifying multiple FOCUS data sources in one MODIFY request.
- The COMPILE command, for translating MODIFY requests into compiled code ready for execution.
- The ACTIVATE and DEACTIVATE statements, for activating and deactivating fields.
- The Checkpoint and Absolute File Integrity facilities, for protecting FOCUS data sources from system failures.
- The ECHO facility, for displaying the logical structure of MODIFY requests.
- Dialogue Manager system variables, which record execution statistics every time a MODIFY request is run.
- FOCUS query commands, which display statistical information on MODIFY request executions and FOCUS data sources.
- COMMIT and ROLLBACK subcommands, for controlling changes made to FOCUS data sources, and for protecting FOCUS data sources from system failures.

All these facilities are described in the sections that follow.

If you are operating in Simultaneous Usage mode (SU), please refer to the appropriate Simultaneous Usage manual.

Modifying Multiple Data Sources in One Request: The COMBINE Command

The COMBINE command allows you to modify two or more FOCUS data sources in the same MODIFY request. The command combines the logical structures of the FOCUS data sources into one structure while leaving the physical structures of the data sources untouched. This combined structure lasts for the duration of the FOCUS session, until you enter another COMBINE command, or it is cleared with the AS CLEAR option. Only one combined structure can exist at a time.

Note the following:

- The combined structure can contain up to 63 segments from up to 63 data sources with one additional reserved for BINS.
- You can only COMBINE FOCUS and XFOCUS data sources with other FOCUS and XFOCUS data sources. You can only COMBINE Fusion data sources with other Fusion data sources.
- You can COMBINE data sources that come from different applications and have different DBA passwords. The only requirement is a valid password for each data source. For more information, refer to the *Describing Data* manual.
- Only the MODIFY and CHECK commands can process combined structures.
- If you are using Simultaneous Usage mode, all the data sources in the combined structure must either be all on the same FOCUS Database Server or all in local mode.
- All MODIFY code compiled in releases prior to 5.2.0 must be re-compiled.
- The differences between JOIN and COMBINE commands are discussed in *Differences Between COMBINE and JOIN Commands* on page 9-197.

Syntax **How to Combine Data Sources**

Enter the COMBINE command at the FOCUS command level (at the FOCUS prompt). The syntax is

```
COMBINE FILES file1 [PREFIX pref1|TAG tag1] [AND]
      .
      .
      .
      filen [PREFIX prefn|TAG tagn] AS asname
```

where:

file1... filen

Are the Master File names for the data sources you want to modify. You can specify up to 63 data sources (you will be limited to fewer data sources if any of these data sources have more than one segment).

pref1... prefn

Are prefix strings for each data source; up to four characters. They provide uniqueness for field names. You cannot mix TAG and PREFIX in a COMBINE structure. See *Referring to Fields in Combined Structures: The PREFIX Parameter* on page 9-194 later in this section.

tag1... tagn

Are aliases for the Master File names; up to eight characters. FOCUS uses the tag name as the qualifier for fields that refer to that data source in the combined structure. You cannot mix TAG and PREFIX in a COMBINE, and you can only use TAG if FIELDNAME is set to NEW or NOTRUNC. See *Referring to Fields in Combined Structures: The TAG Parameter* on page 9-193 later in this section.

AND

Is an optional word to enhance readability.

asname

Is the required name of the combined structure to use in MODIFY procedures and CHECK FILE commands. For example, if you name the combined structure EDJOB, begin the request with:

```
MODIFY FILE EDJOB
```

AS CLEAR

Is the command that clears the combined structure which is currently in effect.

Once you enter the COMBINE command, you can modify the combined structure.

Note:

- TAG and PREFIX may not be used together in a COMBINE.
- You can type the command on one line or on as many lines as you need.

Example **COMBINE Command**

For example, to combine data sources EDUCFILE and JOBFIL, enter:

```
COMBINE FILES EDUCFILE AND JOBFIL AS EDJOB
```

After entering this command, you can run the following request. Notice that the statements pertaining to each data source are placed in different cases (Case Logic is discussed in *Case Logic* on page 9-132). This clarifies the request logic, and makes it easier to understand and clarify the request. The first case modifies the EDUCFILE data source, and the second case modifies the JOBFIL data source.

```
MODIFY FILE EDJOB
PROMPT COURSE_CODE COURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE

CASE EDUCFILE
MATCH COURSE_CODE
  ON MATCH REJECT
  ON MATCH GOTO JOBFIL
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO JOBFIL
ENDCASE

CASE JOBFIL
MATCH JOBCODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
ENDCASE
DATA
```

Syntax **How to Support Long and Qualified Field Names**

If you are using tag names, you must also set the command SET FIELDNAME to NEW or NOTRUNC. The SET FIELDNAME command enables you to activate long (up to 66 characters) and qualified field names. The syntax for this SET command is

```
SET FIELDNAME = type
```

where:

type

Is one of the following:

OLD specifies that 66-character and qualified field names are not supported; the maximum length is 12 characters.

NEW specifies that 66-character and qualified field names are supported; the maximum length is 66 characters. NEW is the default value.

NOTRUNC prevents unique truncations of field names and supports the 66-character maximum.

When the value of FIELDNAME is changed within a FOCUS session, COMBINE commands are affected as follows:

- When you change from a value of OLD to a value of NEW, all COMBINE commands are cleared.
- When you change from a value of OLD to NOTRUNC, all COMBINE commands are cleared.
- When you change from a value of NEW to OLD, all COMBINE commands are cleared.
- When you change from a value of NOTRUNC to OLD, all COMBINE commands are cleared.

Other changes to the FIELDNAME value do not affect COMBINE commands.

Note: For more information on the SET FIELDNAME command, refer to the *Developing Applications* manual.

Reference Referring to Fields in Combined Structures: The TAG Parameter

For a MODIFY request to refer to transaction fields in a combined structure by their transaction field names, the field names must be unique; that is, the transaction field names in one data source cannot appear in other data sources. Refer to any transaction field names that are not unique by their aliases, or use the TAG parameter in the COMBINE command to assign a tag name to the data sources that share the transaction field names.

When a data source has a tag, refer to its transaction field names by affixing the tag name to the beginning of each field name.

For example, this COMBINE command combines data sources EDUCFILE and JOBFIL into the structure EDJOB, and assigns the tag AAA to all the transaction fields in the EDUCFILE data source:

```
COMBINE FILES EDUCFILE TAG AAA AND JOBFIL AS EDJOB
```

When you create a request that modifies this structure, type the EDUCFILE field names with the AAA prefix in front:

```
COMBINE FILES EDUCFILE TAG AAA AND JOBFILE AS EDJOB
MODIFY FILE EDJOB
PROMPT AAA.COURSE_CODE AAA.COURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE
CASE EDUCFILE
MATCH AAA.COURSE_CODE
ON MATCH REJECT
ON NOMATCH INCLUDE
GOTO JOBFILE
ENDCASE
CASE JOBFILE
MATCH JOBCODE
ON MATCH REJECT
ON NOMATCH INCLUDE
ENDCASE
DATA
```

In this request, the tag AAA has been attached to the two transaction field names in the EDUCFILE data source: COURSE_CODE and COURSE_NAME, making the new field names AAA.COURSE_CODE and AAA.COURSE_NAME. Use these tagged field names only in MODIFY requests that modify the combined structure.

Reference Referring to Fields in Combined Structures: The PREFIX Parameter

For a MODIFY request to refer to fields in a combined structure by their field names, the field names must be unique so that there is no ambiguity in the request. That is, the field names in one data source cannot appear in other data sources. If there are field names that are not unique, refer to the fields by their aliases or use the PREFIX parameter in the COMBINE command to assign a prefix of up to four characters to the data sources sharing the field names.

When a data source has a prefix, refer to its field names with the prefix affixed to the beginning of each field name. The field name can be up to 66 characters in length. For example, this COMBINE command combines data sources EDUCFILE and JOBFILE into the structure EDJOB, and assigns the prefix ED to all the fields in the EDUCFILE data source:

```
COMBINE FILES EDUCFILE PREFIX ED JOBFILE AS EDJOB
```

When you enter a request modifying the structure, type the EDUCFILE field names with the ED prefix in front:

```
COMBINE FILES EDUCFILE PREFIX ED JOBFILE AS EDJOB
MODIFY FILE EDJOB
PROMPT EDCOURSE_CODE EDCOURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE

CASE EDUCFILE
MATCH EDCOURSE_COD
    ON MATCH REJECT
    ON NOMATCH INCLUDE
GOTO JOBFILE
ENDCASE

CASE JOBFILE
MATCH JOBCODE
    ON MATCH REJECT
    ON NOMATCH INCLUDE
ENDCASE
DATA
```

In this request, the prefix ED has been attached to the two field names in the EDUCFILE data source: COURSE_CODE and COURSE_NAME. The new field names are EDCOURSE_CODE and EDCOURSE_NAME.

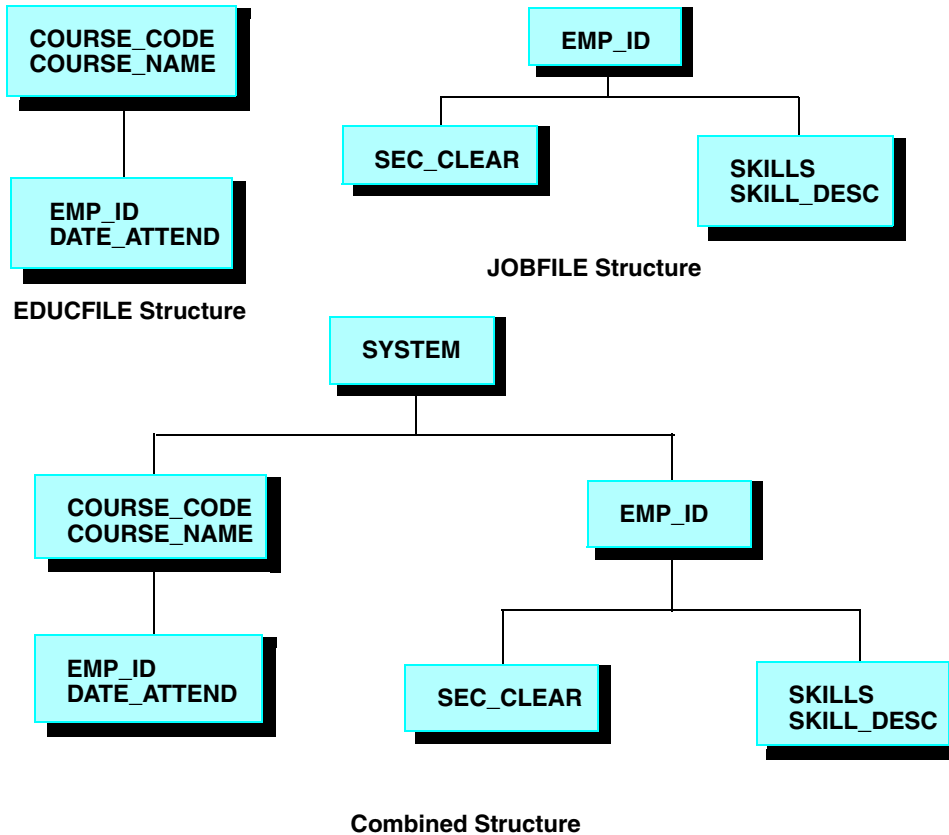
You use these prefixed field names only in MODIFY requests modifying the combined structure. These prefixed field names are not displayed by either the ?F query or the CHECK command.

Note: A MODIFY COMBINE with prefixes cannot be loaded through the LOAD facility. However, the unloaded compiled and uncompiled versions will run. For more information on compiling MODIFY requests see *Compiling MODIFY Requests: The COMPILE Command* on page 9-198. For more information on loading data sources, see the *Developing Applications* manual.

Procedure How Data Source Structures Are Combined

Combined structures start with a dummy root segment called SYSTEM, which becomes the parent of the root segments of the individual data sources. The SYSTEM segment contains no data. This is not an alternate view; the relationships between segments in each data source remain the same.

The following figure shows how two data sources, EDUCFILE and JOBFIL, are combined into one structure. The first two diagrams represent the EDUCFILE and JOBFIL structures; the third diagram represents the combined structure. Note that the relationship between the two segments in each data source does not change.



Field names are considered duplicates when two or more fields are referenced with the same field name or alias. Duplication can occur if a COMBINE is done without a prefix or a tag. Duplicate fields are not allowed in the same segment. The second occurrence is never accessed by FOCUS and the following warning message is generated when CHECK and CREATE FILE are issued:

(FOC1829) WARNING. FIELDNAME IS NOT UNIQUE WITHIN A SEGMENT: *fieldname*

Differences Between COMBINE and JOIN Commands

The COMBINE command differs from the JOIN command in the following ways:

- The JOIN command is effective for TABLE, TABLEF, MATCH, GRAPH, and CHECK commands, but is not effective for MODIFY requests (except for the LOOKUP function). The COMBINE command is effective only for MODIFY requests and CHECK commands.
- The JOIN command joins a variety of FOCUS and non-FOCUS data sources. The COMBINE command combines FOCUS data sources only.
- The JOIN command can only join data sources with common fields. The COMBINE command can combine all FOCUS data sources.
- The JOIN command joins data source structures together at segments with a common field. This can invert some of the segment relationships in the cross-referenced data source (see alternate file view in the *Describing Data* and *Creating Reports* manuals). The COMBINE command combines the data source structures under a dummy root segment. Segment relationships remain intact.

Syntax How to Use the ? COMBINE Query

To display information on the combined structure currently in effect, enter:

```
? COMBINE
```

FOCUS responds

```
FILE=name TAG PREFIX
file-1 tag-1 prefix-1
file-2 tag-2 prefix-2
file-3 tag-3 prefix-3
. . .
. . .
file-n tag-n prefix-n
```

where:

```
name
```

Is the name of the combined structure.

```
file-1 ... file-n
```

Are the names of the data sources that make up the combined structure.

```
tag-1 ... tag-n
```

Are the tags attached to the field names in the data source. These tags correspond to the aliases given to the data source(s) in the combined structure.

```
prefix-1 ... prefix-n
```

Are the prefixes attached to the field names in the data source.

The ? COMBINE query shows up to 63 entries.

For example, when data source EDUCFILE is combined with data source JOBFIL, enter the command

```
? COMBINE
```

to display the following information:

```
COMBINE EDUCFILE AND JOBFIL AS EDJOB
>
? COMBINE
FILE=EDJOB          TAG          PREFIX
      EDUCFILE
      JOBFIL
>
```

Note: TAG and PREFIX may not be mixed in a COMBINE.

Reference **Error Messages for COMBINE**

```
(FOC???) MAXIMUM NUMBER OF 'COMBINES' EXCEEDED. CLEAR SOME AND RE-ENTER:
```

The number of separate COMBINE commands exceeds the current limit of 63.

Compiling **MODIFY** Requests: The **COMPILE** Command

The COMPILE command translates a MODIFY request stored in a procedure into an executable code module. This module, like an object code module, cannot be edited by a user. However, it loads faster than the original request because the MODIFY statements have already been interpreted by FOCUS (the initialization time of a compiled MODIFY module can be four to ten times faster than the original request). Compiling a request can save a significant amount of time if the request is large and must be executed repeatedly. You compile the request once, and execute the module as many times as you need it.

Enter the COMPILE command at the FOCUS command level (the FOCUS prompt). The syntax is

```
COMPILE focexec [AS module]
```

where:

focexec

Is the name of the procedure where the request is stored.

module

Is the name of the module. The default is the procedure name.

Procedure names and module names are system dependent.

To execute a module, enter from the FOCUS command level:

`RUN module`

where *module* is the name of the module. You will see no difference in execution between the module and the original request, but it will load much faster.

Please note the following regarding compilation of MODIFY requests:

- The procedure to be compiled may only contain one MODIFY request. It may not contain any other FOCUS, Dialogue Manager, or operating system statements.
- Before compiling a request or executing a module, allocate all input and output files such as transaction files and log files. These allocations must be in effect at run time.
- Before compilation, issue any SET, USE, COMBINE, or JOIN statements necessary to run the request.
- If the data source you are modifying is joined to another data source (using the JOIN command) during compilation, it must be joined to the data source at run time.
- If you are modifying a combined structure (using the COMBINE command), the structure must be combined both at compilation and at run time.
- Procedures prompt for Dialogue Manager variable values at compilation time. These values cannot be changed at run time.
- If you are using FOCUS security to prevent unauthorized users from executing the request, the password you set at compilation time must be the same one set at run time.

Active and Inactive Fields

This section discusses active and inactive fields. When you run a request, FOCUS keeps track of which transaction fields are active or inactive during execution:

- Active fields have incoming data for them. You may use active fields to add, update, and delete segment instances.
- Inactive fields do not have incoming data for them. You can use inactive fields in calculations only.

When a MATCH statement matches on an inactive field, the request returns to the beginning (the TOP case in case requests) to avoid modifying segments for which data is not present.

If a MATCH or NEXT statement executes an INCLUDE action, all segment instances having active fields are added to the data source.

If a MATCH or NEXT statement executes an UPDATE action, only active fields update the data source. Data source fields corresponding to the inactive incoming fields remain unchanged.

This section covers the following:

- When fields are active and inactive.
- Activating fields with the ACTIVATE statement.
- Deactivating fields with the DEACTIVATE statement.

Reference When Fields Are Active and Inactive

A data field becomes active when:

- It is described in the Master File and it is read in by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement. Note that if the field is declared a conditional field, the following rules apply:
 - In a FIXFORM statement, a conditional field is active when it has a value present in a record.
 - In a CRTFORM, a conditional entry field is active when you enter data for it. A conditional turnaround field is active when you change its value (see Chapter 10, *Designing Screens With FIDEL*).
- The field is assigned a value by a COMPUTE or VALIDATE statement.
- The field is activated by the ACTIVATE statement.

A data field becomes inactive when:

- Execution branches to the top of the request, whether this is done implicitly or by a GOTO statement.
- It modifies a segment instance because of an INCLUDE, UPDATE, or DELETE action.
- It has been made available to the request through the LOOKUP function.
- It is deactivated by the DEACTIVATE statement.

Procedure How to Activate Fields With the ACTIVATE Statement

To activate an inactive field, use the ACTIVATE statement. the ACTIVATE statement performs two tasks:

- It declares a transaction field to be present (considered part of the current transaction). The field can then be used for matching, including, and updating.
- It equates the value of the transaction field to the corresponding data source field. This occurs when both of the following conditions are true:
 - The ACTIVATE statement either appears within or it follows a MATCH or NEXT statement that modifies the segment containing the corresponding data source field.
 - The ACTIVATE statement converts the field from being inactive to active. Included are fields for which the request has not read any data or assigned a value with a compute statement. Fields already active are excluded.

If one of these conditions is not true, the activate statement does not change the value of the field. If the field has no data, FOCUS sets the value of the field to blank if alphanumeric, zero if numeric, and the missing data symbol if the field is described by the MISSING=ON attribute in the Master File (discussed in the *Describing Data* manual).

The syntax of the ACTIVATE statement is

```
ACTIVATE [RETAIN|MOVE] [SEG.]field1 field2 ... fieldn
```

where:

RETAIN

Is an option that activates the field but leaves its value unchanged, even if the ACTIVATE statement converts the field from being inactive to active.

MOVE

Is an option that activates the field and equates its value to the corresponding data source field, even if the field was already active before the ACTIVATE statement.

field1 ...

Are the names of the fields you want to activate. To activate all the fields in one segment, specify any segment field with the prefix SEG. affixed in front of the field name. For example:

```
ACTIVATE SEG.SKILLS
```

This sample request illustrates how ACTIVATE statements affect the fields they specify. The numbers on the margin refer to the notes below. The request is:

```
MODIFY FILE EMPLOYEE

1. FREEFORM EMP_ID CURR_SAL ED_HRS

2. ACTIVATE DEPARTMENT
   MATCH EMP_ID
   ON MATCH REJECT
3.   ON NOMATCH INCLUDE
4. GOTO NEXT_EMP1

   CASE NEXT_EMP1
5. NEXT EMP_ID
   ON NONEXT GOTO EXIT
6.   ON NEXT ACTIVATE RETAIN CURR_SAL DEPARTMENT
7.   ON NEXT UPDATE DEPARTMENT ED_HRS
8.   ON NEXT GOTO NEXT_EMP2
   ENDCASE

   CASE NEXT_EMP2
9. NEXT EMP_ID
   ON NONEXT GOTO EXIT
10.  ON NEXT ACTIVATE CURR_SAL DEPARTMENT ED_HRS
11.  ON NEXT ACTIVATE MOVE CURR_SAL
12.  ON NEXT GOTO NEXT_EMP3
   ENDCASE

   CASE NEXT_EMP3
13. NEXT EMP_ID
   ON NONEXT GOTO EXIT
14.  ON NEXT UPDATE CURR_SAL DEPARTMENT ED_HRS
   ENDCASE

DATA
EMP_ID=222333444, CURR_SAL=50000, ED_HRS=40, $
END
```

When you run the request, the following happens:

1. The request reads the record:

```
EMP_ID=222333444, CURR_SAL=50000, ED_HRS=40, $
```

2. The statement

```
ACTIVATE DEPARTMENT
```

activates the DEPARTMENT field. Since the request did not read any data for this field and the statement precedes the MATCH and NEXT statements, FOCUS equates the field value to blank.

The transaction record is as follows:

```
Transaction Record:
```

```
EMP_ID: 222333444 (active)
CURR_SAL: 50000 (active)
ED_HRS: 40 (active)
DEPARTMENT: blank (active)
```

3. The MATCH statement does not find the EMP_ID value in the data source. It therefore includes the record in the data source as a new segment instance. All fields included in the instance, EMP_ID, CURR_SAL, DEPARTMENT and ED_HRS, become inactive.
4. The request branches to the NEXT_EMP1 case.
5. The request moves the current position in the data source to the next segment instance after EMP_ID 444. This instance contains the following fields:

```
Database Segment Instance:
```

```
EMP_ID: 326179357
CURR_SAL: 21780.00
ED_HRS: 75.00
DEPARTMENT: MIS
```

6. The statement

```
ACTIVATE RETAIN CURR_SAL DEPARTMENT
```

activates the CURR_SAL and DEPARTMENT fields. The RETAIN keyword prevents their values from changing. The transaction record is now:

```
Transaction Record:
```

```
EMP_ID: 326179357 (inactive)
CURR_SAL: 50000 (active)
DEPARTMENT: blank (active)
ED_HRS: 40 (inactive)
```

7. The statement

```
UPDATE DEPARTMENT ED_HRS
```

changes the DEPARTMENT field value in the segment instance to blank and deactivates the DEPARTMENT field on the transaction record. Since the ED_HRS transaction field is inactive, it does not change the data source ED_HRS value. The segment instance is now:

Database Segment Instance:

```
EMP_ID: 326179357  
CURR_SAL: 21780.00  
DEPARTMENT: blank  
ED_HRS: 75.00
```

The request did not use the CURR_SAL transaction field to update the instance, so the CURR_SAL field remains active. The transaction record is as follows:

Transaction Record:

```
EMP_ID: 326179357 (inactive)  
CURR_SAL: 50000 (active)  
DEPARTMENT: BLANK (inactive)  
ED_HRS: 40 (inactive)
```

- 8.** The request branches to the NEXT_EMP2 case.
- 9.** The request moves the current position to the next current instance after EMP_ID 326179357. This instance contains the following fields:

Database Segment Instance:

```
EMP_ID: 451123478  
CURR_SAL: 16100.00  
DEPARTMENT: PRODUCTION  
ED_HRS: 50.00
```

10. The statement

```
ACTIVATE CURR_SAL DEPARTMENT ED_HRS
```

declares the CURR_SAL, DEPARTMENT, and ED_HRS transaction fields to be active. Since CURR_SAL was already active, its value does not change. DEPARTMENT and ED_HRS are converted into active fields, and their values change to that of the DEPARTMENT and ED_HRS fields in the segment instance. The transaction record is now:

```
Transaction Record:
```

```
EMP_ID: 451123478 (inactive)
CURR_SAL: 50000 (active)
DEPARTMENT: PRODUCTION (active)
ED_HRS: 50 (active)
```

11. The statement

```
ACTIVATE MOVE CURR_SAL
```

declares the CURR_SAL transaction field to be active. The MOVE keyword changes the value of CURR_SAL to that of the CURR_SAL field in the segment instance, even though the CURR_SAL field was already active. The transaction record is now:

```
Transaction Record:
```

```
EMP_ID: 451123478 (inactive)
CURR_SAL: 16100.00 (active)
DEPARTMENT: PRODUCTION (active)
ED_HRS: 50 (active)
```

12. The request branches to the NEXT_EMP3 case.

13. The request moves the current position to the next current instance after EMP_ID 451123478. This instance contains the following fields:

```
Database Segment Instance:
```

```
EMP_ID: 543729165
CURR_SAL: 9000.00
DEPARTMENT: MIS
ED_HRS: 25.00
```

14. The request updates the data source CURR_SAL, DEPARTMENT, and ED_HRS fields using the transaction record, causing the CURR_SAL, DEPARTMENT, and ED_HRS transaction fields to become inactive. The segment instance is now:

Database Segment Instance:

```
EMP_ID: 543729165
CURR_SAL: 16100.00
DEPARTMENT: PRODUCTION
ED_HRS: 50.00
```

The transaction record is now:

Transaction Record:

```
EMP_ID: 543729165 (inactive)
CURR_SAL: 16100.00 (inactive)
DEPARTMENT: PRODUCTION (inactive)
ED_HRS: 50 (inactive)
```

Syntax

How to Deactivate Fields With the DEACTIVATE Statement

To deactivate a field, use the DEACTIVATE statement. If the field is a transaction field, the DEACTIVATE statement changes its value to blank if alphanumeric, zero if numeric, or the MISSING symbol for fields described by the MISSING=ON attribute (discussed in the *Describing Data* manual). It also deactivates the corresponding data source field. The RETAIN option leaves the transaction value unchanged.

The syntax is

```
DEACTIVATE [RETAIN] [SEG.] field-1 field-2 ... field-n
DEACTIVATE [RETAIN] ALL
DEACTIVATE COMPUTES
DEACTIVATE INVALID
```

where:

RETAIN

Is an option that deactivates data source fields but does not change the value of the corresponding transaction fields to blank or 0.

field-1 ...

Are the fields you want to deactivate. To deactivate all the fields in one segment, specify any segment field with the prefix seg. affixed in front of the field name. For example:

```
DEACTIVATE SEG.SKILLS
```

ALL

Is an option that deactivates all fields (including temporary fields) and automatically invokes the INVALID option if the request contains CRTFORM statements (see below).

COMPUTES

Is an option that deactivates all temporary fields.

INVALID

Is an option that causes the following: if the user enters a value on a CRTFORM screen and the value fails a validation test, FIDEL does not redisplay the CRTFORM screen to reprompt the user for a valid value. Rather, it displays the next screen.

Use the INVALID option only with requests containing CRTFORM statements.

The ACTIVATE and DEACTIVATE statements can stand by themselves or they can form part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. These are some sample statements:

ACTIVATE RETAIN SKILLS

ON MATCH DEACTIVATE ALL

ON NONEXT ACTIVATE FULL_NAME SEG.SKILLS JOBS_DONE

Protecting Against System Failures

FOCUS provides three ways to protect your data if your system experiences hardware or software failure while you are executing a MODIFY request. They are:

- The Checkpoint facility.
- The Absolute File Integrity feature.
- The COMMIT and ROLLBACK subcommands.

Syntax

How to Safeguard Transactions With the Checkpoint Facility

The Checkpoint facility limits the number of transactions lost if the system fails when you are modifying a data source. You can set checkpoints for transactions that are being read from a data source, or from the terminal.

When a MODIFY request is executed, it does not write transactions to the data source immediately; rather, it collects them in a buffer. When the buffer is full, FOCUS writes all transactions in the buffer to the data source at one time. This cuts down on the input/output operations that FOCUS must perform. If, however, the system crashes, the transactions collected in the buffer may be lost.

You may cause FOCUS to write more frequently to the data source by using the checkpoint facility. When you activate the Checkpoint facility, FOCUS writes to the data source whenever a pre-determined number of transactions accumulates in the buffer. The point at which FOCUS writes the transactions is called the checkpoint.

You control the Checkpoint facility with the following MODIFY statement

```
CHECK {ON|OFF|n}
```

where:

ON

activates the Checkpoint facility. FOCUS writes to the data source when the buffer accumulates 500 transactions in CMS, or 1000 transactions in MVS.

OFF

deactivates the Checkpoint facility.

n

Activates the Checkpoint facility. FOCUS writes to the data source when the buffer accumulates *n* transactions.

Note that if you set *n* to a smaller number, fewer transactions are processed between checkpoints. This causes FOCUS to perform more input/output operations, thereby decreasing efficiency.

If the system does fail while you are modifying a FOCUS data source, enter the ? FILE query when the system comes back. Look at the number in the bottom row in the right-most column. This is the number of transactions written to the data source by the MODIFY request that was executing when the system came down. You can have the request start processing the transaction data source at the next transaction by using the START command, described in *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* on page 9-57.

The following MODIFY request sets the checkpoint at every tenth transaction:

```
MODIFY FILE EMPLOYEE
CHECK 10
MATCH EMP_ID
PROMPT EMP_ID CURR_SAL
      ON MATCH UPDATE CURR_SAL
      ON NOMATCH REJECT
DATA
```

Reference Safeguarding FOCUS Data Sources: Absolute File Integrity

The Absolute File integrity feature completely safeguards the integrity of a FOCUS data source that you are modifying, even if the system experiences hardware or software failure. When you are using this feature, FOCUS does not overwrite the data source on disk; rather, it writes the changes to another section of the disk. If the request finishes normally, the new section of the disk becomes part of the data source. If the system fails, the original data source is preserved.

Reference **Safeguarding Transactions: COMMIT and ROLLBACK Subcommands**

To use COMMIT and ROLLBACK you must use Absolute File Integrity (see *Managing MODIFY Transactions: COMMIT and ROLLBACK* on page 9-214). Unlike the CHECK statement, COMMIT gives you control over the content of data source changes and ROLLBACK enables you to cancel changes before they have been written to the data source. In case of system failure, COMMIT and ROLLBACK ensure that either all or no transactions are processed.

You can use either COMMIT and ROLLBACK, or the CHECK statement in your MODIFY procedures. If the MODIFY procedure uses COMMIT and ROLLBACK, CHECK processing is not used (see *Managing MODIFY Transactions: COMMIT and ROLLBACK* on page 9-214).

Displaying MODIFY Request Logic: The ECHO Facility

The ECHO facility displays the logical structure of MODIFY requests. This is a good debugging tool for analyzing a MODIFY request, especially if the logic is complex and MATCH and NEXT defaults are being used.

Each ECHO display lists:

- The cases, if case logic is used.
- The MODIFY statements used, such as COMPUTE, VALIDATE, TYPE, GOTO, and IF.
- Each segment modified or used to establish a current position.
- The actions the request takes for ON MATCH, ON NOMATCH, ON NEXT, and ON NONEXT conditions when it is modifying the segment, whether these actions are specified by the request or are by default. Default actions are discussed in *The MATCH Statement* on page 9-59.
- The number of data source fields, the total number of fields (including internal fields), and the total size of the field areas.

To use the ECHO facility, first allocate the ECHO terminal output to ddname HLIPRINT. Then, begin the MODIFY command this way

```
MODIFY FILE file ECHO
```

where *file* is the name of the data source. When you run the request, the request does not modify the data source; rather, the ECHO facility displays the listing at the terminal.

The ECHO facility can store the listing in a file rather than display it on the screen. To do this, allocate the file to ddname HLIPRINT. A record length of 80 bytes is sufficient.

The listing has the form

```
MODIFY ECHO FACILITY  
ECHO OF PROCEDURE: focexec
```

```
-----  
CASE casename  
-----
```

```
statements
```

```
                SEGMENT: segname
```

```
ON MATCH                ON NOMATCH  
-----                -----  
match-actions        nomatch-actions0
```

```
NUMBER OF DATABASE FIELDS : n  
TOTAL NUMBER OF FIELDS   : n  
TOTAL SIZE OF FIELD AREAS : n
```

where:

focexec

Is the name of the procedure that the request is stored in. If you entered the request from a terminal, this line is omitted.

casename

Is the name of the case, if the request uses Case Logic.

statements

Are the MODIFY statements used. (**Note:** MATCH statements are shown separately.)

segname

Is the name of the segment being modified or used to establish a current position.

match-actions

Are actions taken on an ON MATCH or ON NEXT condition, including default actions.

nomatch-actions

Are actions taken on an ON NOMARCH or ON NONEXT condition, including default actions.

n

Is an integer.

NUMBER OF DATABASE FIELDS

Is the number of fields described by the Master File, including fields in cross-referenced segments.

TOTAL NUMBER OF FIELDS

Is the sum of the number of data source fields in the Master File and temporary fields in the MODIFY request. This includes fields automatically created by FOCUS (these fields are listed in *Computing Values: The COMPUTE Statement* on page 9-93).

TOTAL SIZE OF FIELD AREAS

Is the sum of the sizes of data source fields in the Master File and temporary fields in the MODIFY request, measured in bytes.

If you are executing a no-case procedure, the ECHO display lists the names of all segments in the data source. Those segments that you did not use in your request are listed with both MATCH and NOMATCH conditions as REJECT.

A sample request running the ECHO facility is shown below:

```
MODIFY FILE EMPLOYEE ECHO
PROMPT EMP_ID
GOTO SALENTY

CASE SALENTY
MATCH EMP_ID
  ON MATCH PROMPT CURR_SAL
  ON MATCH VALIDATE
    SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
  ON INVALID TYPE
    "SALARY TOO HIGH. PLEASE REENTER THE SALARY"
  ON INVALID GOTO SALENTY
  ON MATCH UPDATE CURR_SAL
ENDCASE
DATA
```

Advanced Facilities

When you run this request, the following display appears. Note that although the request did not specify an ON NOMATCH phrase in the SALENTY case, the ECHO display lists the REJECT action under the ON NOMATCH column for the SALENTY case, because REJECT is the default action for an ON NOMATCH condition.

```
EMPLOYEE FOCUS A1 ON 07/18/2003 AT 10.48.21
```

```
MODIFY ECHO FACILITY
ECHO OF PROCEDURE: MOD76
```

```
-----
CASE TOP
-----
```

```
PROMPT
GOTO SALENTY
```

```
-----
CASE SALENTY
-----
```

```
SEGMENT: EMPINFO
-----
```

MATCH	NOMATCH
-----	-----
PROMPT	REJECT
VALIDATE	
INVALID TYPE	
INVALID GOTO SALENTY	
UPDATE	

```
END OF ECHO:
```

```
NUMBER OF DATABASE FIELDS : 34
TOTAL NUMBER OF FIELDS    : 36
TOTAL SIZE OF FIELD AREAS : 371
```

Dialogue Manager Statistical Variables

After you run a FOCUS request, FOCUS automatically records statistics about the execution in specially designated Dialogue Manager variables. Since these variables do not receive values until after execution is completed, they are not useful in the requests themselves. However, you may use them in procedures after execution (that is, after the Dialogue Manager -RUN control statement).

The variables that pertain to MODIFY requests are:

&TRANS	Number of transactions processed.
&ACCEPTS	Number of transactions accepted into the data source.
&INPUT	Number of segment instances added to the data source.
&CHNGD	Number of segment instances updated.
&DELTD	Number of segment instances deleted.
&DUPLS	Number of transactions rejected because of an ON MATCH REJECT condition.
&NOMATCH	Number of transactions rejected because of an ON NOMATCH REJECT condition.
&INVALID	Number of transactions rejected because transaction values failed validation tests.
&FORMAT	Number of transactions rejected because of format errors.
&REJECT	Number of transactions rejected for other reasons.

For instructions on how to use Dialogue Manager variables to build procedures, see the *Developing Applications* manual.

MODIFY Query Commands

Four query commands display information regarding the MODIFY command and the maintenance of FOCUS data sources. These are:

? COMBINE	Displays information on combined structures (see <i>Modifying Multiple Data Sources in One Request: The COMBINE Command</i> on page 9-190).
? FDT	Displays information regarding the physical attributes of FOCUS data sources (see the <i>Developing Applications</i> manual).
? FILE	Displays information regarding the number of segment instances in FOCUS data sources and the dates and times the data sources were last modified (see the <i>Developing Applications</i> manual).
? STAT	Displays statistics regarding the last execution of a request (see the <i>Developing Applications</i> manual).

Managing MODIFY Transactions: COMMIT and ROLLBACK

COMMIT and ROLLBACK are two MODIFY subcommands. COMMIT gives you control over the content of data source changes and ROLLBACK enables you to undo changes before they become permanent.

The COMMIT subcommand safeguards transactions in case of a system failure and provides greater control (than the MODIFY Checkpoint facility) over which transactions are written to the data source.

The MODIFY CHECK statement only enables you to control the number of transactions that must occur before changes are written to the data source. When using CHECK, you cannot change the checkpoint setting once the MODIFY request begins execution. Similarly, changes cannot be canceled (see *How to Safeguard Transactions With the Checkpoint Facility* on page 9-207 for more information on the CHECK statement).

COMMIT enables you to make changes based on the content of the transactions as well as the number. Changes you do not want to make can be canceled with ROLLBACK, unless a COMMIT has been issued for those changes. Should the system fail, either all or none of your transactions will be processed.

Absolute File Integrity is required in order to use COMMIT and ROLLBACK. Absolute File Integrity is provided automatically by CMS for all data sources, except those that have an A6 file mode. Absolute File Integrity for data sources with an A6 file mode is provided by the FOCUS Shadow Writing Facility. Absolute File Integrity for data sources in MVS/TSO is provided solely by the FOCUS Shadow Writing Facility.

Reference The COMMIT and ROLLBACK Subcommands

The COMMIT and ROLLBACK subcommands are automatically activated in FOCUS and cannot be deactivated. Therefore, unless you omit these subcommands from your code, COMMIT and ROLLBACK processing takes place. If you would rather use CHECK processing, make sure you do not include COMMIT and ROLLBACK subcommands, as they will take precedence over CHECK processing.

Reference Coding With COMMIT and ROLLBACK

COMMIT and ROLLBACK each process a logical transaction. A logical transaction is a group of data source changes in the MODIFY environment that you want to treat as one. For example, you can handle multiple records displayed on a CRTFORM and then processed using the REPEAT command as a single transaction. A logical transaction is terminated by either COMMIT or ROLLBACK. COMMIT and ROLLBACK also can be used for single-record processing.

When COMMIT ends a logical transaction, it writes all changes to the data source. COMMIT can be coded as a global subcommand or as part of MATCH or NEXT logic. The possible MATCH and NEXT statements are:

```
COMMIT
ON MATCH COMMIT
ON NOMATCH COMMIT
ON MATCH/NOMATCH COMMIT
ON NEXT COMMIT
ON NONEXT COMMIT
```

When ROLLBACK ends a logical transaction, it does not write changes to the data source. The ROLLBACK subcommand cancels changes made since the last COMMIT. ROLLBACK cannot cancel changes once a COMMIT has been issued for them.

ROLLBACK can also be coded as a global subcommand or as part of MATCH or NEXT logic. Possible MATCH and NEXT statements are:

```
ROLLBACK
ON MATCH ROLLBACK
ON NOMATCH ROLLBACK
ON MATCH/NOMATCH ROLLBACK
ON NEXT ROLLBACK
ON NONEXT ROLLBACK
```

If the COMMIT fails for any reason (for example, system failure, lack of disk space), no changes are made to the data source. In this way, COMMIT is an all-or-nothing feature that ensures data source integrity.

In the following example, a user may COMMIT or ROLLBACK changes after each group of three records has been processed, or delay the COMMIT subcommand until later by selecting the option to add more records. Changes are stored permanently in the data source when the user chooses to commit the changes or when the procedure is terminated without issuing a ROLLBACK subcommand.

Note: In the following example the COMMIT and ROLLBACK subcommands are included in Case COMM and Case ROLL, respectively.

```
MODIFY FILE EMPLOYEE
COMPUTE ANSWER/A1=;
CRTFORM LINE 1
"ENTER UP TO 3 NEW EMPLOYEES"
" "
"  EMPLOYEE ID    LAST NAME    FIRST NAME"
"1. <EMP_ID(1)    <LAST_NAME(1)  <FIRST_NAME(1) "
"2. <EMP_ID(2)    <LAST_NAME(2)  <FIRST_NAME(2) "
"3. <EMP_ID(3)    <LAST_NAME(3)  <FIRST_NAME(3) "
GOTO MATCHIT

CASE MATCHIT
REPEAT 3
  MATCH EMP_ID
  ON NOMATCH INCLUDE
  ON MATCH REJECT
ENDREPEAT
GOTO DECIDE
ENDCASE

CASE DECIDE
CRTFORM LINE 10
"WHAT WOULD YOU LIKE TO DO NOW? <ANSWER"
" C TO COMMIT CHANGES SO FAR"
" R TO ROLLBACK CHANGES"
" A TO ADD MORE EMPLOYEES"
IF ANSWER EQ 'C' PERFORM COMM
  ELSE IF ANSWER EQ 'R' PERFORM ROLL
  ELSE IF ANSWER EQ 'A' GOTO TOP
  ELSE PERFORM BADCHOICE;
GOTO TOP
ENDCASE

CASE COMM
COMMIT
ENDCASE

CASE ROLL
ROLLBACK
ENDCASE
```

```

CASE BADCHOICE
TYPE "PLEASE ENTER C, R, OR A."
GOTO DECIDE
ENDCASE

DATA
END

```

MODIFY Syntax Summary

This section presents a summary of MODIFY command syntax. The syntax of each statement is shown as part of a MODIFY request. The rest of the summary shows:

- The syntax of the transaction statements FIXFORM, FREEFORM, and PROMPT. The syntax of the CRTFORM statement is shown in Chapter 10, *Designing Screens With FIDEL*.
- The actions you can use in MATCH and NEXT statements.

MODIFY Request Syntax

The following is the syntax of MODIFY requests:

```

MODIFY FILE filename [ECHO|TRACE]
TYPE [ON ddname] [AT START|AT END]
" text "
COMPUTE
field/format = ;
***** transaction subcommand *****
VALIDATE
field = expression ;
    ON INVALID {GOTO ... |PERFORM ... |TYPE [ON ddname] }
    " text "
COMPUTE
field/format = expression ;
MATCH { * [KEYS] [SEG.n] | [WITH-UNIQUES] keyfield(s) [field ... field] }
    ON MATCH action
    ON MATCH action
    .
    .
    ON NOMATCH action
    ON NOMATCH action
    .
    .
    ON MATCH/NOMATCH action

```

MODIFY Syntax Summary

```
REPEAT [*|number] [TIMES] [MAX maximum] [NOHOLD]
  statements
  HOLD [SEG.]field [field ... field]
ENDREPEAT

ACTIVATE [RETAIN|MOVE] [SEG.]field ... field

DEACTIVATE {[RETAIN] [SEG.] field ... field | [RETAIN]
ALL|COMPUTES|INVALID}

CASE casename

GOTO {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}

PERFORM {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}

IF expression
[THEN] {GOTO|PERFORM} {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}
[ELSE {GOTO|PERFORM} {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}]

HOLD [SEG.]field [field ... field]

GETHOLD

NEXT field
  ON NEXT action
  ON NEXT action
  .
  .
  ON NONEXT action
  ON NONEXT action
  .
  .
ENDCASE

COMMIT

ROLLBACK

LOG {TRANS|ACCEPTS|DUPL|NOMATCH|INVALID|FORMAT} [ON ddname]
[MSG {ON|OFF}]

CHECK {ON|OFF|n}

START n

STOP n

DATA {ON ddname|VIA proname}

[END]
```

Transaction Statement Syntax

The following is the syntax for three transaction statements: FIXFORM, FREEFORM, and PROMPT. For CRTFORM syntax, see Chapter 10, *Designing Screens With FIDEL*.

The syntax of the FIXFORM statement:

```
FIXFORM {FROM master|
        [ON ddname] field/[C]format field/[C]format ... [Xn] [X-n]}
```

The syntax of the FREEFORM statement:

```
FREEFORM [ON ddname] field field field ...
```

The syntax of the PROMPT statement:

```
PROMPT {*|field[.text.] field[,text,] . . .}
```

MATCH and NEXT Statement Actions

This section lists the actions that can be taken by MATCH and NEXT statements. They are placed in ON MATCH, ON NOMATCH, ON NEXT, and ON NONEXT phrases. These actions are:

```
ACTIVATE
COMMIT
COMPUTE
CONTINUE (ON MATCH and ON NEXT only)
CONTINUE TO (ON MATCH and ON NEXT only)
CRTFORM
DEACTIVATE
DELETE (ON MATCH and ON NEXT only)
FIXFORM
FREEFORM
GOTO
HOLD
IF
INCLUDE
PERFORM
PROMPT
REJECT
REPEAT (ON MATCH and ON NEXT only)
ROLLBACK
TED (ON MATCH and ON NOMATCH ON NEXT and ON NONEXT)
TYPE
UPDATE (ON MATCH and ON NEXT only)
VALIDATE
```

MODIFY Syntax Summary

The following actions can be used in ON MATCH/NOMATCH phrases:

ACTIVATE
COMMIT
CRTFORM
DEACTIVATE
GOTO
HOLD
IF
PERFORM
PROMPT
ROLLBACK
TED

The following actions can be used in ON INVALID phrases:

GOTO
PERFORM
TYPE

CHAPTER 10

Designing Screens With FIDEL

Topics:

- Introduction
- Describing the CRT Screen
- Using FIDEL in MODIFY
- Using FIDEL in Dialogue Manager
- Using the FOCUS Screen Painter

FIDEL, the FOCUS Interactive Data Entry Language, enables you to design full-screen forms for data entry and application development. You use FIDEL both with MODIFY for building data maintenance and inquiry screens, and with Dialogue Manager for building applications that accept values for variables at run time.

Introduction

Describing the CRT Screen on page 10-7 describes the facilities of FIDEL that are common to both MODIFY and Dialogue Manager. This introduction explains how MODIFY facilities and FIDEL interact, and describes the FIDEL facilities that are specific to MODIFY. *Using FIDEL in Dialogue Manager* on page 10-78 describes the interaction between Dialogue Manager and FIDEL.

From the FOCUS TED editor, you can also use the FOCUS Screen Painter with both MODIFY and Dialogue Manager to interactively build and view screens online. With the Screen Painter, you design the layout of the form and the Screen Painter automatically generates the FIDEL code to build it. The FOCUS Screen Painter is described in *Using the FOCUS Screen Painter* on page 10-84.

The two simple examples on the following pages demonstrate how to generate a screen form by using the CRTFORM and -CRTFORM syntax. Note how closely FIDEL syntax resembles TABLE syntax for creating headings.

Note: FIDEL only supports fixed format records with LRECL=80.

Using FIDEL With MODIFY

The following example of a simple MODIFY CRTFORM illustrates the use of FIDEL with the resulting screen (the numbers refer to the explanation and are *not* part of the code):

```
MODIFY FILE EMPLOYEE
1. CRTFORM
2.   "EMPLOYEE UPDATE"
3.   "EMPLOYEE ID #: <EMP_ID   LAST NAME:  <LAST_NAME"
4.   "DEPARTMENT: <DEPARTMENT SALARY:    <CURR_SAL"
5. MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
6. DATA
   END
```


This request sets up a form to update the last name, department and current salary. Processing is as follows:

1. CRTFORM generates the visual form and invokes FIDEL. The form begins on line one of the screen unless specified otherwise with the LINE option (see *Using Multiple CRTFORMs: LINE* on page 10-51).
2. Each line on the screen begins and ends with double quotation marks. This is a line of text that serves as a title. Note the close correspondence to the syntax used to create headings in a TABLE request.
3. The second screen line specifies two data fields: EMP_ID and LAST_NAME. A data entry field is indicated by a left caret, followed by the field name or alias from the Master File. The text, EMPLOYEE ID #: and LAST NAME: identifies each field on the screen. This informs the operator where to enter the data.
4. This is the last line within double quotation marks. It signals the end of the CRTFORM. In this case it identifies and defines two more data fields: DEPARTMENT and CURR_SAL. When you run the MODIFY request, the form instantly appears on the screen:

```

EMPLOYEE UPDATE
EMPLOYEE ID #:  LAST NAME:
DEPARTMENT:   SALARY:

```

The number of characters allotted for each data entry field on the screen defaults to the display format for that particular field in the Master File. You can optionally specify a format for screen display that is shorter than the default.

The operator can now fill in the data entry areas with the appropriate information.

5. The request continues with MODIFY MATCH logic.
6. This line tells FOCUS that the incoming data is from the terminal. In conjunction with CRTFORM, it implies full-screen data input. You can also use DATA VIA FIDEL or, in CMS, DATA VIA FI3270.

When you use FIDEL with MODIFY, you are setting up full-screen forms for the maintenance of data source fields. Most MODIFY features, such as conditional and non-conditional fields, automatic application generation, Case Logic, multiple record processing, error handling, validation tests, logging transactions, and typing messages to the terminal, work with FIDEL.

With MODIFY you also have access to additional screen control options such as clearing the screen, specifying and changing the size of the screen, and designating the particular line on which the form starts.

Using FIDEL With Dialogue Manager

The following example of a simple -CRTFORM illustrates the use of FIDEL in Dialogue Manager and the resulting screen (the numbers refer to the explanation and are *not* part of the code):

```
1. -CRTFORM
2. -"MONTHLY SALES REPORT FOR <&CITY/10"
3. -"BEGINNING PRODUCT CODE IS: <&CODE1/3"
   -"ENDING PRODUCT CODE IS: <&CODE2/3"
4. -"REGIONAL SUPERVISOR IS: <&REGIONMGR/5"
   TABLE FILE SALES
   HEADING CENTER
   "MONTHLY REPORT FOR &CITY"
   "PRODUCT CODES FROM &CODE1 TO &CODE2"
   " "
   SUM UNIT_SOLD AND RETURNS AND COMPUTE
   RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
   BY PROD_CODE
   IF PROD_CODE IS-FROM &CODE1 TO &CODE2
   FOOTING CENTER
   "REGIONAL SUPERVISOR: &REGIONMGR"
   END
```

The procedure sets up a form for gathering run-time variables for a TABLE request: &CITY, the city for the report; &CODE1 and &CODE2, a range of product codes; and ®IONMGR, the regional supervisor. Processing is as follows:

1. -CRTFORM generates the visual form, invokes FIDEL, and clears the screen.
2. Each line on the screen begins with a dash and double quotation marks ("-"), and ends with double quotation marks. Note this first line of the screen form contains text and a variable field, &CITY, which has a length of 10. This specifies ten spaces on the screen for entering the value. The data entry field is indicated by the left caret.
3. The next few lines of the screen form contain both text and variable fields with formats.
4. The last line within double quotation marks signals the end of the -CRTFORM. When the FOCEXEC executes, the screen displays the following form:

```
MONTHLY SALES REPORT FOR
BEGINNING PRODUCT CODE IS:
ENDING PRODUCT CODE IS:
REGIONAL SUPERVISOR IS:
```

The operator can now fill in values for the run-time variables. After the operator transmits the screen by pressing Enter, the values entered on the screen are sent to the variables. The regular FOCUS commands are stacked and executed when the end of the procedure is reached.

When you use FIDEL with Dialogue Manager, you can define input fields as amper variables that receive values at run time to adjust to specific processing requirements. Because they are *not* data fields and are not part of the Master File, they do not automatically have a format. You must allocate space for them on the screen. You can do this directly on the -CRTFORM as in the previous example, or through a -SET statement.

Dialogue Manager supports two additional control statements: -CRTFORM BEGIN and -CRTFORM END. The statement -CRTFORM BEGIN signals the beginning of the screen form. You can then enter screen lines as well as other Dialogue Manager control statements. You then signal the end of the screen form with the statement -CRTFORM END. This allows you to use Dialogue Manager statements between screen lines while building the form.

Screen Management Concepts and Facilities

The following briefly outlines the FIDEL capabilities that are common to both MODIFY and Dialogue Manager and defines the common terminology:

- The MODIFY CRTFORM statement and the Dialogue Manager -CRTFORM control statement both automatically invoke FIDEL. All succeeding lines placed within double quotations make up the actual screen form. Note the common syntax between TABLE headings (see the *Creating Reports* manual) and CRTFORM screen lines.
- You can combine a CRTFORM and a -CRTFORM in one procedure. However, they must remain within their own environments. The MODIFY CRTFORM contains data source fields, whereas the Dialogue Manager -CRTFORM contains amper variables.
- The term *field* in this chapter refers to either a data source field name in conjunction with MODIFY or an amper variable in conjunction with Dialogue Manager.
- You can define a CRTFORM in MODIFY or a -CRTFORM in Dialogue Manager that has more lines than on your CRT screen. FIDEL provides scrolling capabilities.
- It is important to note the difference between the physical screen on the terminal and the logical CRTFORM or form. A form generated by one CRTFORM or -CRTFORM statement can take up many screens or less than one screen.
- You can specify three types of fields on the screen: input, display only, and turnaround (both display and update). Data entry and turnaround fields are considered unprotected areas on the screen because you may input values or replace what is there. Display values are considered protected areas on the screen because you cannot alter what is there (see *Data Entry, Display and Turnaround Fields* on page 10-14).
- You can set PF key controls and specify cursor positioning. You can specify screen attributes such as background effects, highlighting, and color to enhance readability of the screen. You can also change screen attributes depending on the outcome of various tests (see *Controlling the Use of PF Keys* on page 10-20, *Specifying Screen Attributes* on page 10-25, and *Using Labeled Fields* on page 10-28).

Note: This chapter is written specifically for the IBM 3270 terminal, which supports PF key and cursor control, scrolling and screen attributes. Examples use the program syntax for the CMS operating system.

Using FIDEL Screens: Operating Conventions

The following procedures apply for filling in *all* FIDEL screens:

- To move from field to field, press the Tab key. You can also move the cursor around the screen using the arrow keys.
- When filling in values on the screen, you may use any of the keys on the keyboard. Some terminals automatically prevent the entry of a non-numeric character in a field identified as computational.
- To scroll forward or backward through a long CRTFORM (from screen to screen) press the PF8 or PF7 key, respectively (or PF20, PF19).
- To transmit the screen, press the Enter key.
- If you make an error, the transaction may not be transmitted and an error message may appear at the bottom of the screen. You can correct the error and retransmit the screen.
- To signal the end of data entry, press the PF3 or PF15 key or type END in an unprotected area. In MODIFY, this terminates the request. In Dialogue Manager, this terminates the FOCEXEC procedure.

The following operating procedures are specific to MODIFY:

- To return to the first screen without transmitting the current screen, press the PF2 key or the key set to QUIT.
- If the screen clears at any time, press the Enter key to bring it back.

Note: The PF key settings referred to here are the default settings. Any PF key can be redefined using the SET statement.

Describing the CRT Screen

The MODIFY statement CRTFORM or the Dialogue Manager control statement -CRTFORM, followed by the screen layout, generates a form. Within one MODIFY procedure, you can use an unlimited number of screen lines (within memory constraints). Each screen line can contain a maximum of 78 characters of text and data.

In MODIFY, you can use up to 255 CRTFORM statements in a procedure. In Dialogue Manager, there is no limit to the number of -CRTFORM statements that you may use in one procedure.

All the basic options described here can be used with both MODIFY and Dialogue Manager. Options that are specific to MODIFY are discussed in *Using FIDEL in MODIFY* on page 10-41 and those specific to Dialogue Manager are discussed in *Using FIDEL in Dialogue Manager* on page 10-78.

The following example shows the syntax of a simple MODIFY CRTFORM using the LOWER case option, followed by two screen lines containing various screen elements: text, a spot marker, and a field (numbers refer to the explanation; they are *not* part of the code):

```
1. CRTFORM LOWER
2. "PLEASE FILL IN THE EMPLOYEE ID # </1"
3. "EMPLOYEE ID #: <EMP_ID"
   MATCH EMP_ID
   .
   .
   .
```

Processing is as follows:

1. CRTFORM invokes FIDEL and generates the form. The LOWER case option specifies that what is entered from the terminal in lowercase will remain in lowercase.
2. The first line of the screen contains descriptive text.
 </1 is a spot marker which skips one blank line.
3. The last line of the screen contains two screen elements: descriptive text that identifies the field and the data source field EMP_ID. The last line between quotation marks signals the end of the CRTFORM.

The form generated appears as follows:

<pre>PLEASE FILL IN THE EMPLOYEE ID # EMPLOYEE ID #:</pre>
--

Specifying Elements of the CRTFORM

To create the visual form, you enter the screen lines one after the other within double quotation marks. For each screen line, you can specify various screen elements such as descriptive text and fields. A left caret (<) followed by the name of the field generates the position where data is to be entered onto the screen.

You may need to use two FOCEXEC lines to describe one physical CRTFORM line. Simply omit the double quotation marks (") at the end of the first line and omit them at the beginning of the next line as well. Everything between the set of double quotation marks will read as one screen line on the CRTFORM.

Syntax Invoking FIDEL: CRTFORM and -CRTFORM

The following is a summary of the complete syntax for generating a CRTFORM in MODIFY or a -CRTFORM in Dialogue Manager. The individual options and screen elements are described in detail in specific sections later in the chapter. The syntax is

```
[ - ] CRTFORM [ option option... ]  
[ - ] "screen element [screen element....]"
```

where:

[-] CRTFORM

Automatically invokes FIDEL and sets up the visual form. Subsequent lines describe the screen.

option option...

Refers to screen control options. (See *Using FIDEL in MODIFY* on page 10-41 and *Using FIDEL in Dialogue Manager* on page 10-78.)

[-] "screen element.."

Can be user-defined text, fields, or spot markers. Spot markers define the next place on the screen where a screen element will appear. Both spot markers and fields are preceded by a left caret and optionally closed by a right caret (see *Specifying Elements of the CRTFORM* on page 10-8).

Note:

- You can create simple screen forms by typing the FIDEL code into your procedures with your text editor. However, it is easier to build more complex forms using many screen attributes and field labels using the FOCUS Screen Painter.
- You can use the asterisk (*) with CRTFORM in FIDEL to generate a CRTFORM containing all of the data source's fields automatically (that is, without specifying individual fields). See *Generating Automatic CRTFORMs* on page 10-47 for information on CRTFORM *, its syntax and variations.
- Do not begin any field used in a CRTFORM or FIXFORM statement with Xn , where n is any numeric value. This applies to fields in the Master File and computed fields.

Defining a Field

Labels, prefixes, attributes, and formats are parts of the definition of a particular field. In Dialogue Manager, the first character is an ampersand, which signals an amper variable. (The entire definition is preceded by a left caret and optionally closed by a right caret.)

Note: Fields with a text (TX) format cannot be used in CRTFORM or -CRTFORM. However, they can be entered interactively using TED (see *Entering Text Data Using TED* in Chapter 9, *Modifying Data Sources With MODIFY*, for using text fields in MODIFY).

Syntax

How to Define a Field in FIDEL

The syntax for defining a field is as follows.

In MODIFY:

```
<[:label.] [prefix.] [attribute.] field[/length] [>]
```

In Dialogue Manager:

```
<[&:label.] [prefix.] [attribute.]&variable[/length] [>]
```

where:

:label. | *&:label.*

Is a user-defined label of up to 12 characters associated with a field. It may not contain embedded blanks (see *Using Labeled Fields* on page 10-28).

prefix.

Refers to D. or T., which designate a display or turnaround field, respectively (see *Data Entry, Display and Turnaround Fields* on page 10-14).

attribute.

Is the abbreviation or full name of a screen attribute (see *Specifying Screen Attributes* on page 10-25).

field

Is the name of the field or variable being defined.

&variable

Is for data entry. Can be a data source field or a temporary field.

/length

Is the length of the field as it appears on the screen. In MODIFY, you need to define a length only if you want the screen length to be different from the format length that is defined in the MASTER or COMPUTE. In Dialogue Manager, you need to define a length only if not previously defined.

Note: When you use the abbreviations for attributes, you do not need to use the dot separator between attributes or between a prefix and an attribute (see *Specifying Screen Attributes* on page 10-25).

Example **Defining a Field**

The following is an example of the syntax of a Dialogue Manager screen line defining the variable field &CITY:

```
-CRTFORM  
- "<&:L01.T.HIGH.&CITY/7"  
.  
.  
.
```

The elements on the second line which define the variable field &CITY are:

1. The left caret generates a place for the variable on the screen.
2. &:L01 is a label that identifies the data entry area on the screen (see *Using Labeled Fields* on page 10-28).
3. T. is a prefix that defines the variable as a turnaround field. If the variable has been given a value within the FOCEXEC, it is displayed. Otherwise a default value is displayed. The operator can then change the value.
4. .HIGH. is a screen attribute specifying that the contents of the field will be highlighted.
5. &CITY/7 is the name of the variable field with a length specification. The specified length is seven characters. That is, the space that will be allotted on the screen for input of data is seven characters long.

Prefixes, labels, and screen attributes are explained fully in *Data Entry, Display and Turnaround Fields* on page 10-14, *Specifying Screen Attributes* on page 10-25, and *Using Labeled Fields* on page 10-28.

Reference **Difference in FIDEL When Used With MODIFY and Dialogue Manager**

The following chart outlines the similarities and differences of FIDEL when used with MODIFY and Dialogue Manager:

MODIFY	Dialogue Manager
<code>CRTFORM [options]</code>	<code>-CRTFORM [options]</code>
UPPER/LOWER CLEAR/NOCLEAR WIDTH/HEIGHT TYPE LINE	UPPER/LOWER BEGIN/END TYPE
<code>"screen elements"</code> <code>text</code> <code><spot marker[>]**</code> <code><field/length[>]*</code> <code>prefix.(D. or T.)***</code> <code>attribute.</code> <code>:label.</code>	<code>"screen elements"</code> <code>text</code> <code><spot marker[>]**</code> <code><field/length[>]**</code> <code>prefix.(D. or T.)***</code> <code>attribute</code> <code>&:label.</code>

* The right caret denotes a non-conditional field.

** The right caret has no meaning, but may be used for increased clarity.

*** Prefixes, attributes and labels are part of the definition of the field on the screen. They do not stand alone.

Using Spot Markers for Text and Field Positioning

Because the lengths of fields vary, text does not automatically align uniformly on the screen. Spot markers are available to help you position both text and fields. Please note that a spot marker is essential to eliminate trailing blanks at the end of the first line, if your screen line description takes up two FOCEXEC lines.

The syntax and usage of the different spot markers are shown in the following chart:

Marker	Example	Usage
<code><n or <n></code>	<code><50</code>	Positions the next character in column 50.
<code><+n or <+n></code>	<code><+4</code>	Positions the next character four columns from the last non-blank character.
<code><-n or <-n></code>	<code><-1</code>	Positions the next character one column to the left of the last character. This marker's function is to suppress or write over the attribute byte at the beginning and the end of a field.
<code></n or </n></code>	<code></2</code>	Positions the next character at the beginning of the line that is two lines from the last (skips two lines). Note: The last line is blank and is created when a double quotation mark (") is encountered.
<code><0X or <0X></code>	<code><0X</code>	Positions the next character immediately to the right of the last character (skip zero columns). This is used to help position data on a FIDEL screen when a single screen line is coded as two lines in a FOCEXEC. No spaces are inserted between the spot marker and the start of a continuation line (see Note 3 in the following example).

Note: You can optionally use the right caret >. This is useful when the next character in the line is a left caret. It also enhances readability.

Suppose you want the various input data fields arranged across the screen in vertical sections, left justified, and in horizontal segments marked off with lines. Using spot markers, you can create the desired screen as shown in the following example:

```

MODIFY FILE EMPLOYEE
CRTFORM
  "EMPLOYEE UPDATE"
1.  "</1"
    "-----"
    "EMPLOYEE ID #: <EMP_ID   LAST NAME: <LAST_NAME"
1.  "</1"
2.  "DEPARTMENT: <DEPARTMENT <+3 CURRENT SALARY:<0X>
    <CURR_SAL"
    "-----"
    "BANK: <BANK_NAME"
    "-----"
MATCH EMP_ID
.
.
.
DATA
END

```

The spot markers in the example perform the following functions:

1. </1 generates a blank line.
2. <+3 moves the word CURRENT three spaces to the right of the last letter in the word DEPARTMENT. <0X> skips no spaces. No extra spaces are inserted between this and the next word (<CURR_SAL) on the continuation line. There is, in fact, one space before the field which is an attribute byte that marks the start of a field.

The screen appears as:

```

EMPLOYEE UPDATE

-----

EMPLOYEE ID #:   LAST NAME:

DEPARTMENT:   CURRENT SALARY:
-----

BANK:
-----

```

Specifying Lowercase Entry: UPPER/LOWER

All text that is entered from the terminal is normally translated to uppercase letters. You can override this default and preserve both uppercase and lowercase text by using the lowercase option. The syntax is

```
[ - ] CRTFORM [UPPER|LOWER]
```

where:

UPPER

Translates all characters to uppercase. This is the default.

LOWER

Reads lowercase data from the screen. Once you specify LOWER, every screen thereafter is a lowercase screen until you specify UPPER.

Note: In MODIFY, when you use multiple CRTFORMs on the same screen (using LINE n), you can mix UPPER and LOWER among the forms.

Data Entry, Display and Turnaround Fields

There are three types of data or variable fields that can be specified on the CRTFORM: data entry, display, and turnaround.

You can also compute data fields (see *Computing Values: The COMPUTE Statement* in Chapter 9, *Modifying Data Sources With MODIFY*, for rules about computing data fields) and specify them as entry, display, or turnaround on the CRTFORM. You can convert a turnaround field to a display field dynamically.

In MODIFY, fields can also be designated as conditional or unconditional (see *Conditional and Non-Conditional Fields* on page 10-42). We recommend that for data entry, you use conditional fields (left caret only) so that the values in your data source are not replaced by a blank or a zero if you do not enter data for the field.

For most turnaround fields, we recommend that you use non-conditional fields (both carets). A non-conditional turnaround field remains active whether you enter data or not. Because the value in the data source is displayed in the field, that value remains in the data source if you do not change it. Because the field remains active, the values for your VALIDATEs and COMPUTE s are then accurate (see *Conditional and Non-Conditional Fields* on page 10-42 for a complete explanation of the use of conditional and non-conditional fields in MODIFY).

The following outlines the rules for specification of different types of fields.

Syntax **Data Entry (for Data Entry Only)**

In MODIFY, the syntax is

```
<field[/length] [>]
```

where:

```
<field[>]
```

Is the name of the field. Reserves space on the screen for data entry into that field and does not display the current value of the field.

In MODIFY, if only the left caret is used, data entry is conditional. If both carets are used, the field is non-conditional (see *Conditional and Non-Conditional Fields* on page 10-42).

In Dialogue Manager the syntax is

```
<&variable[/length] [>]
```

where:

```
<&variable[>]
```

Is the name of the variable field. Reserves space on the screen for data entry into that field and does not display the current value of the field.

In Dialogue Manager, the option of the right caret is meaningless. Usually for the FOCEXEC to run, you must supply a value for each variable. If you do not, FOCUS assumes a blank or a 0 for that value.

Syntax **Display (for Information Only)**

Data is displayed in a protected area and cannot be altered.

In MODIFY, the syntax is

```
<D. field[/length]
```

In Dialogue Manager, the syntax is

```
<D. &variable[/length]
```

where:

```
D.
```

Is the prefix placed in front of a field, indicating that the data or value is to be displayed. The current value of the field appears on the screen, but in a protected area which cannot be changed.

Note that the right caret is meaningless for display fields.

Syntax **Turnaround Field (for Display and Change)**

Data is displayed in an unprotected area and can be altered.

In MODIFY, the syntax is:

`<T.field[/length] [>]`

In Dialogue Manager, the syntax is:

`<T.&variable[/length] [>]`

where:

T.

Is the prefix placed in front of a field to indicate that it is a turnaround field. The current value of the field is displayed on the screen. However, the operator may change the value, as it is not in a protected area.

In MODIFY, if only the left caret is present, the T. field is treated as conditional. If the right caret is used, the field is non-conditional, and the value is treated as present, even if unchanged (see *Conditional and Non-Conditional Fields* on page 10-42).

In Dialogue Manager, the changed value for the turnaround variable field will substitute everywhere in the FOCEXEC where it is subsequently encountered.

Note: In MODIFY, in order to display data from a data source field or present it for turnaround, a position in the data source must first be established through the use of a MATCH or NEXT statement, or value must be assigned in a COMPUTE. A computed field cannot be set and displayed in the TOP case, where data entry is processed prior to computations. For example, one of the phrases

`ON MATCH CRTFORM`

`ON NEXT CRTFORM`

must be used. A position is thus established in the data source, and the values of the fields in existing records are now available for display as protected or unprotected fields.

You can also match on a key field and go to a case (see *CRTFORMs and Case Logic* on page 10-57) in which you display a CRTFORM using display and turnaround fields.

Example Using Data Entry, Display, and Turnaround Fields With MODIFY

The following example combines two CRTFORMs in a single MODIFY request and shows the use of entry, display and turnaround fields (numbers refer to the explanation below; they are *not* part of the code):

```

MODIFY FILE EMPLOYEE
1. CRTFORM
   "ENTER EMPLOYEE ID#: <EMP_ID>"
   "PRESS ENTER"
   "</2>"
2. MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH CRTFORM
   " "
   "REVISE DATA FOR SALARY AND DEPARTMENT"
   "ENTER NEW DATA FOR EDUCATION HOURS"
   " "
3.   "EMPLOYEE ID #: <D.EMP_ID   LAST_NAME: <D.LAST_NAME>"
   " "
4.   "SALARY:      <T.CURR_SAL>"
   "DEPARTMENT: <T.DEPARTMENT>"
5.   "EDUCATION HOURS: <ED_HRS>"
   ON MATCH UPDATE CURR_SAL DEPARTMENT ED_HRS
DATA
END

```

The procedure matches the employee ID, displays both the ID and the last name, and then displays the current salary and department for turnaround. Education hours is a data entry field.

Note that when the procedure executes, both CRTFORMs are displayed immediately. However, the display and turnaround fields in the second CRTFORM do not display data until the operator fills in the first form and presses Enter. We therefore recommend you use the LINE option.

When a FORMAT ERROR occurs, all data entered up to that point is processed and cannot be changed in the course of your transaction.

The processing is as follows:

1. CRTFORM generates the first form which begins on line 1 (the second CRTFORM is displayed, but without values):

```
ENTER EMPLOYEE ID #:  
PRESS ENTER  
  
REVISE DATA FOR SALARY AND DEPARTMENT  
ENTER NEW DATA FOR EDUCATION HOURS  
  
EMPLOYEE ID #:   LAST NAME:  
SALARY:  
DEPARTMENT:  
EDUCATION HOURS:
```

2. The procedure continues with the MATCH logic. If the ID number that is input matches an ID in the data source, the display and turnaround fields on the second CRTFORM display the data. Assume the operator enters 818692173 and presses Enter.

The following is displayed:

```
ENTER EMPLOYEE ID #: 818692173  
PRESS ENTER  
  
REVISE DATA FOR SALARY AND DEPARTMENT  
ENTER NEW DATA FOR EDUCATION HOURS  
  
EMPLOYEE ID #: 818692173   LAST NAME: CROSS  
SALARY:      27062.00  
DEPARTMENT:  MIS  
EDUCATION HOURS:
```

3. This screen line contains two display fields.
4. The next two screen lines contain turnaround fields.
5. The last line is a data entry field.

Note: To display fields from a unique segment, the ON MATCH CONTINUE TO, ON NEXT, or MATCH WITH-UNIQUES phrase must have been executed (see *Modifying Data: MATCH and NEXT* in Chapter 9, *Modifying Data Sources With MODIFY*).

In Dialogue Manager, in order to display values with D. or T., a value must have been supplied for the variable prior to the initiation of the -CRTFORM. System variables are an exception to this rule, as the system automatically supplies their values.

Computed fields in both MODIFY and Dialogue Manager can be displayed in any kind of CRTFORM.

Example Using Data Entry, Display, and Turnaround Fields With Dialogue Manager

The following example illustrates the use of D. fields and system variables in a Dialogue Manager -CRTFORM:

```

1.  -SET &CITY = STAMFORD;

2.  -CRTFORM
3.  -"YEARLY SALES REPORT FOR <T.&CITY/10"
4.  -"DATE: <D.&DATE  TIME: <D.&DATEMDYY"
    -" "
    -"ENTER BEGINNING PRODUCT CODE RANGE: <&BEGCODE/3"
    -"ENTER ENDING PRODUCT CODE RANGE: <&ENDCODE/3"
    -"ENTER NAME OF REGIONAL SUPERVISOR: <&REGIONMGR/15"

TABLE FILE SALES
HEADING CENTER
"YEARLY REPORT FOR &CITY"
"PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
" "
SUM UNIT_SOLD AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
IF CITY EQ &CITY
FOOTING CENTER
"REGION MANAGER: &REGIONMGR"
"CALCULATED AS OF &DATE"
END

```

The example processes as follows:

1. The -SET sets a default value for &CITY:

```
FOR WHICH CITY DO YOU WANT A REPORT?
```

2. -CRTFORM generates the screen form:

```
YEARLY SALES REPORT FOR STAMFORD
DATE: 02/22/2003  TIME: 13.42.38

ENTER BEGINNING PRODUCT CODE RANGE:
ENTER ENDING PRODUCT CODE RANGE:
ENTER NAME OF REGIONAL SUPERVISOR:

```

3. The transaction value for &CITY is Stamford, the value that was previously supplied in the -SET statement.
4. Note that the variables &DATE and &DATEMDYY are system variables. The values are supplied by the system and displayed on the form.

Controlling the Use of PF Keys

The terminal operator can use certain PF keys to control the execution of a FIDEL application. Normally, the following keys are used:

- PF3 and PF15 mean END and terminate execution.
- PF2 means Cancel and cancels the transaction in MODIFY.
- PF7 and PF8 page Backward and Forward respectively.

Note: All other keys return the value of the PF key when pressed.

Several facilities are available to assist you in controlling various screen operations:

- You can reset PF key functions. You can also set PF keys to branch to particular cases in MODIFY or labels in Dialogue Manager.
- You can set the cursor on a specified position on the screen (see *Specifying Cursor Position* on page 10-33).
- You can use the cursor position on the screen to perform a branch or action based on a test (see *Determining Current Cursor Position for Branching Purposes* on page 10-36).

Reference Default Settings for PF Keys

The default PF key settings are as follows:

PF Key	Function
PF01	HX
PF02	CANCEL
PF03, PF15	END
PF04, PF16	RETURN
PF05, PF17	RETURN
PF06, PF18	RETURN
PF07, PF19	BACKWARD
PF08, PF20	FORWARD
PF09, PF21	RETURN
PF10, PF22	RETURN
PF11, PF23	RETURN
PF13	RETURN
PF12, PF24	UNDO
PF14	RETURN

You can display the current PF key settings by issuing the FOCUS query command:

? PFKEY

This displays a formatted table of all the current values.

Resetting PF Key Controls

You can reset PF key functions in FIDEL for both CRTFORMs and -CRTFORMs using the FOCUS SET command with the following syntax

```
SET PFxx = function
```

where:

xx

Is a one or two-digit PF key number.

function

Is one of the following:

END in MODIFY, exits the procedure; in Dialogue Manager, is equivalent to QUIT. That is, END exits the procedure.

CANCEL in MODIFY, cancels the transaction and returns to the TOP case. Do not use the CANCEL setting in Dialogue Manager.

FORWARD pages forward.

BACKWARD pages backward.

RETURN has no specific screen action. Returns the PF key name in the PFKEY field because it is not yet defined. To set the PFKEY field, use COMPUTE in MODIFY or -SET in Dialogue Manager.

HELP displays text supplied with the HELPMESSAGE attribute for any field on the MODIFY CRTFORM. Position the cursor on the data entry area of the desired field, and press the PF key you have defined for HELP. If no help message exists for that field, the following message is displayed:

```
NO HELP AVAILABLE FOR THIS FIELD.
```

The following example sets the PF03 key for paging backward and the PF04 key for paging forward:

```
SET PF03=BACKWARD, PF04=FORWARD
```

Note: When changing PF key settings, make sure that at least one key is set to END. If you set a PF key to FORWARD, you should also set one to BACKWARD.

Setting PF Key Fields for Branching Purposes

You can create a menu of processing options. The operator can then indicate a choice by pressing a particular PF key. To assign a specific processing function to a PF key, you must specify a field named PFKEY. Which PF key the operator presses determines the value of the PFKEY field.

You can use the PF keys designated as Return keys, as well as the Enter key. You define a variable called PFKEY (in MODIFY) or &PFKEY (in Dialogue Manager) and then test its value after the CRTFORM is displayed. Which branch takes place depends on which PFKEY the operator presses.

In MODIFY, the syntax is

```
COMPUTE
PFKEY/A4=;
```

where:

```
PFKEY/A4
```

Is a four-character field, whose value is determined by which key the operator presses at run time.

In Dialogue Manager, the syntax is

```
-SET &PFKEY='  ';
```

where:

```
&PFKEY
```

Is a four-character field, whose value is determined by which key the operator presses at run time.

```
= '  ';
```

Is the allocation of four character spaces for the field.

The following example shows how PF keys can be tested in MODIFY:

```
1. COMPUTE
   PFKEY/A4=;
2. CRTFORM
   "SELECT OPTION"
   "INPUT   PRESS PF4"
   "UPDATE  PRESS PF5"
   "DELETE  PRESS PF6"
3. IF PFKEY EQ 'PF04' GOTO INCASE
   ELSE IF PFKEY EQ 'PF05' GOTO UPCASE
   ELSE IF PFKEY EQ 'PF06' GOTO DELCASE
   ELSE GOTO TOP;
   .
   .
   .
```

The example processes as follows:

1. The COMPUTE statement specifies a four-character field PFKEY.
2. CRTFORM generates the form which supplies the operator with three options:

```
SELECT OPTION
INPUT   PRESS PF4
UPDATE  PRESS PF5
DELETE  PRESS PF6
```

3. The IF test determines what case to branch to depending on the value of the PFKEY field. For example, if the operator presses PF4, the value for PFKEY is PF04, and the request branches to an input case INCASE.

Specifying Screen Attributes

Screen attributes (such as highlighting, colors, and so on) can be applied to the fields on the CRTFORM and the -CRTFORM. They can also be used as background effects and can be applied to the fields depending on the result of tests.

The following attributes are available on 3270 IBM terminals:

Function	Abbreviation	Short Name
Flash or Blink	F	FLAS or BLIN
Underline	U	UNDE
Invert or Reverse Video	I	INVE or REVV
Clear*	C	CLEA
Blue	B	BLUE
Red	R	RED
Pink	P	PINK
Green	G	GREE
Aqua	A	AQUA
Turquoise	T	TURQ
Yellow	Y	YELL
White	W	WHIT
Nodisplay*	N	NODI
Return to default	\$	\$
Highlight or Intensify*	H	HIGH or INTE

Note:

- *Clear, Nodisplay, and Highlight or Intensify can be used on all terminals. Clear also sets the highlight off for entry and turnaround fields. Nodisplay is not supported for D. or T. fields. The remaining attributes are also known in the FOCUS community as extended attributes.
- Use of abbreviations is recommended, except for TURQ.

When an attribute is unsupported on a particular terminal or is specific to a version of FOCUS under another operating system, the attribute is ignored. Therefore, there is no need for code changes between terminals and/or operating systems.

To use the screen attributes other than C, N, and H you must notify FOCUS that your terminal is equipped to display them. Issue the FOCUS SET command:

```
SET EXTTERM=ON
```

This allows a procedure to be operated on a variety of terminals. FOCUS automatically detects a 3279 model terminal and sets EXTTERM to ON by default.

If your terminal does not properly recognize extended attributes, due to a "terminfo" compatibility problem, stray characters will appear on your screen. You may turn off extended attribute recognition with the command:

```
SET EXTTERM=OFF
```

Programs with extended attributes and EXTTERM=OFF will run as if extended attributes had not been coded in the program.

Make sure that your terminal has the extended attribute options needed before you turn EXTTERM on. There are many different IBM 3270 models. Generally, the color terminals in the 3279 series have most of the options. However, even if a terminal has the physical capability to support all of the attributes, it may be defined to the operating system as a lower grade terminal. In such cases, you must ascertain whether or not all the attributes can be used.

The syntax for defining screen attributes in MODIFY is

```
<[:label] [.attribute.] field[>]
```

The syntax for defining screen attributes in Dialogue Manager is

```
<[&:label] [.attribute.] &variable[>]
```

where:

.attribute.

Is one or more of the attributes. Note the dots (periods) before and after each attribute or entry in an attribute list.

field

Names the field to which the attributes apply.

&variable

Names the variable field to which the attributes apply.

Note: Labels and their use are discussed in *Using Labeled Fields* on page 10-28.

The following chart shows you how to use these attributes in conjunction with prefixes (D. and T.), where X is the name of a field or variable:

<code>.HT.X</code>	Highlighted T.
<code>.CT.&X</code>	Unhighlighted T.
<code>.N.X</code>	Nodisplay entry, (for example, for passwords)
<code>.H.&X</code>	Highlighted entry
<code>.C.X</code>	Unhighlighted entry
<code>.HD.X</code>	Highlighted D.

The following usage considerations apply when using screen attributes:

- An attribute stays in effect until another attribute changes it.
- A list of attributes may be composed entirely of abbreviations in any order. If abbreviations only are used, you do not need the dot separator between attributes.
- The last mentioned option in a group of mutually exclusive attributes will be taken.
- A color or flash overrides a highlight, clear, or Nodisplay.
- If short names are used, the first four letters identify the attribute. Each name must be separated by a dot. Either abbreviations or short names can be used, but they cannot be mixed without a dot separator.
- Full names may be used as well. Each must be delimited by a dot.
- You can change screen attributes during the course of a terminal session by using labeled fields.

Note the following examples:

<code>.AID.</code>	Aqua inverted display field.
<code><.RED.FLASH.</code>	Red flashing field.
<code><.RED.FLAS.</code>	Red flashing field.
<code><.PIN.</code>	Inverted pink field (color overrides).
<code><I.YELL.</code>	Inverted yellow field.

Using Background Effects

If a field is absent, the attribute affects the protected portion of the screen; that is, the text. Both a beginning and ending dot as well as a space between the attribute and the text are needed. For example:

```
"<.RED. ENTER EMP_ID:"
```

This will print the words ENTER EMP_ID: in red. Note the space between .RED. and ENTER EMP_ID:. A right caret may also be inserted for clarity.

The line:

```
"<.INVE.RED. <.CLEAR.EMP_ID"
```

will turn the background color to red. CLEAR changes the background for the input field EMP_ID back to black.

An attribute stays in effect until another attribute changes it on a physical screen. Therefore, if <.INVE.RED. is in the upper left corner, the entire screen will be in inverse red unless some other background attribute is provided later. In the example above, the <.CLEAR is used to limit the effect to one area.

Note: .CLEAR. and .HIGH. only work when they are used in conjunction with a field. They do not work alone or simply with text.

Using Labeled Fields

You can use labels to identify a specific field on the screen. They are necessary to perform the following functions:

- Dynamically change screen attributes during processing depending on the results of tests.
- Position the cursor on the screen, or read the position of the cursor on the screen, where there is no pre-existing field.

The syntax for a labeled field in MODIFY is

```
<:label.field
```

The syntax for a labeled field in Dialogue Manager is

```
<&:label.&variable
```

where:

```
<[&]:label.
```

Is a user-defined label. It starts with a colon (:), and may be up to 66 characters long including the colon. You may not use embedded blanks.

```
field
```

Is any field on the CRTFORM. It can be a field created specifically for appending a label.

```
&variable
```

Is any variable field on the CRTFORM. It can be a field created specifically for appending a label.

The following rules apply:

- A label cannot occur by itself. It must be used with a field.
- A label must be declared using a COMPUTE, -SET, or -DEFAULTS statement.
- Setting a label to \$ returns its field to the default attribute.

Example Using a Labeled Field With MODIFY

For example, in MODIFY:

```
COMPUTE
:ONE/A6='  ' ;
CRTFORM
"<:ONE.EMP_ID"
```

The label :ONE is set to a format of A6 and is the identifier of the field EMP_ID.

Example Using a Labeled Field With Dialogue Manager

For example, in Dialogue Manager:

```
-SET &:ONE='  ' ;
-CRTFORM
-"<&:ONE.&CITY/10"
```

In this Dialogue Manager example, the label &:ONE is set to a format of A4 and is the identifier of the field &CITY.

Note: When you are dealing with many complex labels and attributes, we advise you to use the FOCUS Screen Painter which allows you to do everything without learning the detailed syntax (see *Using the FOCUS Screen Painter* on page 10-84).

Dynamically Changing Screen Attributes

The screen attributes in a FIDEL form can be changed during the course of the terminal session in which they are defined. This allows you to design easy-to-read and easy-to-use procedures. For instance, after an error occurs, you can turn a specific field into flashing red to alert the operator.

The mechanism for changing the attribute is to put a label before the field. Then, issue a COMPUTE in MODIFY, or a -SET in Dialogue Manager, to assign the label new attribute values. When the screen is next displayed, it takes on the characteristics of the provided attributes.

The following example shows how to use a COMPUTE in MODIFY to dynamically change an attribute value:

```
COMPUTE
  :ATTRIB/A12=IF CURR_SAL GT 50000 THEN 'FLASH' ELSE '$';
CRTFORM
  "AMOUNT <:ATTRIB.T.CURR_SAL>"
IF CURR_SAL GT 50000 GOTO TOP ELSE GOTO OTHER;
.
.
.
```

This generates an attribute value for the label ATTRIB. If the CURR_SAL is greater than 50,000, the field will flash; otherwise, it observes the default setting.

The following example shows the use of a -SET statement to assign an attribute value in Dialogue Manager:

```
-SET &AMOUNT=0;
-SET &:ATTRIB='      ';
-TOP
-CRTFORM
-"AMOUNT: <&:ATTRIB.T.&AMOUNT>"
-SET &:ATTRIB=IF &AMOUNT GT 100 THEN 'FLASH' ELSE '$';
-IF &AMOUNT GT 100 GOTO TOP;
.
.
.
```

This generates an attribute value for the label &:ATTRIB, changing &AMOUNT to flashing if the value is greater than 100. Be sure to use -SET to establish the label in the beginning of the procedure.

Note: When you use CRTFORMs in either MODIFY or Dialogue Manager, the labels you assign must precede the fields with which they are associated; labels cannot occur by themselves. Use COMPUTE statements to dynamically change screen text attributes, setting the label equal to the COMPUTE (see previous example).

You can convert a T. field to a D. field dynamically; however, you cannot convert a D. field to a T. field. The method for changing turnaround fields to display fields is the same as that for changing screen attributes dynamically.

```

MODIFY FILE EMPLOYEE
1. CRTFORM
2. "SALARY UPDATE"
2. " "
3. "EMPLOYEE ID #: <.INVE.EMP_ID LAST NAME: <0X
   <.CLEAR.D.LAST_NAME"
4. MATCH EMP_ID
   ON NOMATCH REJECT
5.   ON MATCH CRTFORM LINE 10
6.   ENTER SALARY"
   " "
   "SALARY: <:HERE.T.CURR_SAL>"
7. COMPUTE
   :HERE/A12=IF CURR_SAL GT 100000 THEN 'D' ELSE 'T';
   IF CURR_SAL GT 100000 GOTO TOP;
   ON MATCH UPDATE CURR_SAL
DATA
END

```

This procedure constructs a form to update salaries. It processes as follows:

1. CRTFORM generates the screen form and invokes FIDEL.
2. Provide text for the CRTFORM; empty quotation marks indicate a blank line on the form.
3. The next two lines contain the following screen elements:

```
EMPLOYEE ID #:
```

Is text describing the conditional data field EMP_ID.

```
.INVE.
```

Is a screen attribute that displays the field EMP_ID in reverse video.

```
LAST NAME:
```

Is text describing the field LAST_NAME.

```
.CLEAR.
```

Is a screen attribute that clears the .INVE. attribute, returning the D. (display-only) field LAST_NAME to the default display.

4. The request continues with MODIFY MATCH logic.
5. If EMP_ID matches, another CRTFORM is generated on line 10 of the same screen.

Describing the CRT Screen

- The next three lines contain the following screen elements:

`ENTER SALARY :`

Is text describing the CURR_SAL field.

`" "`

Generates a blank line.

`:HERE`

Is a label identifying the CURR_SAL field.

- This COMPUTE evaluates the field CURR_SAL and defines it as a turnaround (T.) field or a display (D.) field, depending on the value of CURR_SAL. If the salary is greater than 100,000, the field is a display field (and cannot be updated); if the salary is less than 100,000, the field is a turnaround field (and can be updated).

The resulting CRTFORM is as follows:

```
SALARY UPDATE
```

```
EMPLOYEE ID #:      LAST NAME:
```

```
ENTER SALARY
```

```
SALARY:
```

Specifying Cursor Position

To specify cursor position, simply choose the field where you want the cursor positioned. You may specify the field by its field name or by its label. You can set the cursor at a specific place on the screen by computing or setting the value of the field CURSOR (in MODIFY) or &CURSOR (in Dialogue Manager).

The syntax for the field which controls the cursor position in MODIFY is

```
COMPUTE
CURSOR/A66= expression;
```

where:

CURSOR/A66

Is a 66-character alphanumeric field.

expression

Is terminated with a semicolon and can be anything, including the full field name, its full alias, or a unique truncation of either, or the label itself. This determines the position of the cursor.

For example:

```
COMPUTE
CURSOR/A66=IF TESTNAME GT 100 THEN 'EMP_ID'
ELSE 'LAST_NAME' ;
```

The position of the cursor will be on the field EMP_ID if the value of test name is greater than 100, or it will be on the field LAST_NAME if test name is less than or equal to 100.

You may also position the cursor using a field label. For example:

```
COMPUTE
CURSOR/A66=IF TESTNAME GT 100 THEN ':ONE'
ELSE ':TWO' ;
```

Note: If the field name is not unique, FIDEL uses the first occurrence of the field name (going from left to right across each line and then down to the next line) to set or test the cursor position.

In MODIFY, the variable CURSORINDEX can also be used to compute the position of the cursor at a particular record when there are multiple indexed records displayed in a single CRTFORM. This feature is commonly used for placing the cursor on invalid fields after VALIDATE statements. The syntax is

```
COMPUTE  
CURSORINDEX/I5=expression;
```

where:

CURSORINDEX/I5

Is a five-digit integer field. Refers to the current value of the subscript being processed from the CRTFORM.

expression

May be any expression, but in most applications will be set equal to REPEATCOUNT.

Note: See *Case Logic, Groups, CURSORINDEX and VALIDATE* on page 10-65 for a full example of the use of CURSORINDEX using Case Logic, multiple fields and the VALIDATE subcommand. Also, multiple record processing is discussed in full in *Multiple Record Processing* in Chapter 9, *Modifying Data Sources With MODIFY*.

In Dialogue Manager, the syntax for positioning the cursor is

```
-SET &CURSOR=expression;
```

where:

&CURSOR

Is a variable specifically referring to the position of the cursor.

expression

Is terminated with a semicolon and can be any valid expression including the field name or label itself. It determines the position of the cursor.

The following example illustrates the positioning of the cursor on the screen in Dialogue Manager using labeled fields:

```

1.  -SET &:AAA = '      ';
    -SET &:BBB = '      ';
2.  -PROMPT &YR.PLEASE ENTER YEAR NEEDED.
3.  -SET &CURSOR = IF &YR GT 1984 THEN ':AAA' ELSE ':BBB';
    -*
4.  -CRTFORM
    -"MONTHLY REPORT FOR THE CITY <&:AAA.&CITY/10"
    -"YEARLY REPORT FOR THE AREA <&:BBB.&AREA/1"
    .
    .
    .

```

This processes as follows:

1. Two -SET statements declare the labels, which are themselves variables.
2. The -PROMPT statement prompts the operator for a value for &YR.
3. The -SET statement sets an IF test as the value for the variable &CURSOR. If the value of &YR is greater than 1984, the position of the cursor is set to the label :AAA; otherwise, it is set to the label :BBB.
4. If the operator supplies the value 85 for &YR, the visual form generated is as follows, and the cursor is positioned at the variable &CITY:

MONTHLY REPORT FOR THE CITY YEARLY REPORT FOR THE AREA

The remainder of the FOCEXEC might then branch to a TABLE request for a monthly report for that city. Had the year been earlier than 84, the cursor would have been positioned on AREA. The branch might then be to a TABLE request for a yearly report for that area.

Caution: In Dialogue Manager, be sure to set &CURSOR to the label name without the & (ampersand). Use :AAA, not &:AAA.

Determining Current Cursor Position for Branching Purposes

Rather than having the operator type a response, you can create a menu on which you list options. To select an option, the operator moves the cursor to the correct line on the screen and presses the Enter key. FOCUS senses the cursor position and takes action based upon it (such as branching to a particular case or field).

To do this, you must specify a 66 character field that contains the current cursor position, CURSORAT. You may identify a field on the screen by a label or by its field name.

The syntax that defines the field used to read the cursor position in MODIFY is

```
COMPUTE  
CURSORAT/A66=;
```

where:

```
CURSORAT/A66
```

Is the field whose value is determined by the field name, or label of the field, on which the cursor is positioned when the operator presses Enter.

In Dialogue Manager, the syntax is

```
-SET &CURSORAT= '      ' ;
```

where:

```
&CURSORAT
```

Is a variable whose value is determined by the field name, or label of the field, on which the cursor is positioned when the operator presses Enter.

If the actual cursor position is not on any field, the value of CURSORAT is the nearest preceding field. If there are no preceding fields, the value of CURSORAT is the TOP of the CRTFORM. That is, the value is at the very beginning of the CRTFORM.

In the following example, field XYZ is a computed field for the purpose of creating a labeled field wherever necessary on the CRTFORM:

```

MODIFY FILE EMPLOYEE
1. COMPUTE
  CURSORAT/A66=;
2. :ADD/A1=;
  :UPP/A1=;
3. XYZ/A1=;
4. CRTFORM
  "POSITION CURSOR NEXT TO OPTION DESIRED"
  "THEN PRESS ENTER"
  " "
  "<:ADD.XYZ  ADD RECORDS"
  "<:UPP.XYZ  UPDATE RECORDS"
5. IF CURSORAT EQ ':ADD' GOTO ADD ELSE
  IF CURSORAT EQ ':UPP' GOTO UPP ELSE GOTO TOP;

CASE ADD
CRTFORM LINE 1
  "THIS CRTFORM ADDS RECORDS"
  " "
  "EMPLOYEE ID #: <EMP_ID"
  "LAST NAME:    <LAST_NAME"
  "FIRST NAME:   <FIRST_NAME"
  "HIRE DATE:    <HIRE_DATE"
  "DEPARTMENT:   <DEPARTMENT"
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
ENDCASE

CASE UPP
CRTFORM LINE 1
  "THIS CRTFORM UPDATES RECORDS"
  " "
  "EMPLOYEE ID #: <EMP_ID"
  "DEPARTMENT:    <DEPARTMENT"
  "JOB CODE:     <CURR_JOBCODE"
  "SALARY:       <CURR_SAL"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_JOBCODE CURR_SAL
ENDCASE
DATA
END

```

This example processes as follows:

1. The COMPUTE establishes the field CURSORAT.
2. The second and third COMPUTEs declare the labels :ADD and :UPP.
3. The third COMPUTE establishes a field XYZ for the purpose of using labels.
4. CRTFORM generates the following visual form beginning on the first line of the screen:

```
POSITION CURSOR NEXT TO OPTION DESIRED
THEN PRESS ENTER

ADD RECORDS
UPDATE RECORDS
```

5. An IF phrase tests the value of the field CURSORAT. If the operator places the cursor next to ADD RECORDS, the value of CURSORAT is :ADD and a branch to Case ADD will be performed. If the operator places the cursor next to UPDATE RECORDS, the value of CURSORAT is :UPP and Case UPP will be performed.

Annotated Example: MODIFY

The following example illustrates the syntax for a MODIFY CRTFORM using dynamically changing attributes:

```
MODIFY FILE EMPLOYEE
1. CRTFORM
2. "EMPLOYEE UPDATE"
3. "</1"
4. "EMPLOYEE ID #: <.INVE.EMP_ID"
GOTO UPDATE
CASE UPDATE
5. MATCH EMP_ID
ON NOMATCH REJECT
6. ON MATCH CRTFORM LINE 1
" "
7. "LAST NAME: <.INVE.T.LAST_NAME"
"DEPARTMENT: <.CLEAR.T.DEPARTMENT>"
"SALARY: <:ATTRIB.T.CURR_SAL>"
8. ON MATCH COMPUTE
:ATTRIB/A12 = IF CURR_SAL GT 50000 THEN 'FLASH.INVE';
MSG/A60 = IF CURR_SAL GT 50000 THEN 'PLEASE REENTER' ELSE ' ';
ON MATCH TYPE "<MSG"
ON MATCH IF CURR_SAL GT 50000 GOTO UPDATE;
ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
ENDCASE
DATA
END
```

This procedure sets up a form to update the department and current salary. It processes as follows:

1. CRTFORM generates the visual form and invokes FIDEL.
2. This line contains a screen element set between double quotations marks. It is a line of descriptive text.
3. This line contains another screen element, a spot marker that skips one line.
4. These lines contain four screen elements—'EMPLOYEE ID #:' is text describing the field; the field EMP_ID is described as a conditional data entry field. The contents will be displayed in reverse video because .INVE. is a screen attribute defining the field.

The visual form generated is as follows:

```
EMPLOYEE UPDATE
EMPLOYEE ID #: (reverse video)
```

Enter Employee ID # 818692173.

5. The request continues with MODIFY MATCH logic.
6. If the EMP_ID matches, another CRTFORM is generated. It is placed on LINE 1 and thus replaces the previous CRTFORM on the screen.
7. The CRTFORM defines three turnaround fields:

The LAST_NAME field. The .INVE. attribute displays the field in reverse video.

The DEPARTMENT field. The .CLEAR. attribute displays the field in regular video.

The CURR_SAL field. The appearance of the field value depends on the value of the :ATTRIB field. When the CURR_SAL value first appears, the :ATTRIB field is empty and the value appears in regular video. If you enter a CURR_SAL value greater than 50,000, the :ATTRIB field receives the attribute FLASH.INVE, displaying the CURR_SAL value in flashing inverse (or reverse) video. The CRTFORM appears as follows:

```
LAST NAME:   CROSS
DEPARTMENT:  MIS
SALARY:      27062.00
```

8. If the CURR_SAL field value is greater than 50,000 when you press Enter, the COMPUTE statement assigns the :ATTRIB label the FLASH.INVE attribute.
9. If the CURR_SAL field value is greater than 50,000 when you press Enter, the IF statement branches back to the CASE UPDATE statement. This displays the second CRTFORM with the CURR_SAL value in reverse video.

Note: If you are using a terminal emulator you may not be able to view the attribute FLASH.INVE.

Annotated Example: Dialogue Manager

The following sample -CRTFORM illustrates the syntax for dynamic control of attributes in Dialogue Manager:

1. -PROMPT &CITY.FOR WHICH CITY DO YOU WANT A REPORT?.
2. -SET &:ATTRIB = IF &CITY EQ STAMFORD THEN 'INVE' ELSE 'CLEAR';
-*
3. -CRTFORM
4. -"MONTHLY SALES REPORT"
5. -"Date: <D.&DATE Time: <D.&TOD"
6. -"Beginning Code is: <&:ATTRIB.&BEGCODE/3"
- "Ending Code is: <&:ATTRIB.&ENDCODE/3"
- "Regional Supervisor is: <&:ATTRIB.®IONMGR/15"
TABLE FILE SALES
HEADING CENTER
"MONTHLY REPORT FOR &CITY"
"PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
" "
SUM UNIT_SOLD AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
IF CITY EQ &CITY
FOOTING CENTER
"REGION MANAGER: ®IONMGR"
"CALCULATED AS OF &DATE"
7. END

The example processes as follows:

1. The -PROMPT prompts the operator for a value for &CITY.
2. The -SET statement sets the label :ATTRIB to INVE if the city is Stamford, causing each field labeled :ATTRIB in the remainder of the -CRTFORM to be displayed in reverse video.
3. -CRTFORM generates the visual form and invokes FIDEL.
4. The first line of the screen form contains text.
5. The second line contains the current date and time as display fields. Since they are in protected areas of the screen, they cannot be altered.

6. Each of the next three lines contains descriptive text and one field. Each field has a label which displays the field in reverse video if the city is Stamford.

The screen displays the following -CRTFORM:

```
MONTHLY SALES REPORT
Date: 01/08/97      Time: 10:50:16
Beginning Code is:
Ending Code is:
Regional Supervisor is:
```

7. After the operator presses Enter, the values entered in the screen form are sent to the variables. The TABLE request executes when END is encountered.

Using FIDEL in MODIFY

The following standard MODIFY functions and concepts work with FIDEL in the building of CRTFORMs (for additional information on these functions):

- Conditional and non-conditional field specification (see *Conditional and Non-Conditional Fields* on page 10-42).
- The FIXFORM statement which can be used before the first CRTFORM. This enables you to mix data sources (see *Using FIXFORM and FIDEL in a Single MODIFY* on page 10-45).
- Automatic application generation, which enables you to use several simple statements to generate automatic data management procedures and CRTFORMs (see *Generating Automatic CRTFORMs* on page 10-47).
- Multiple CRTFORMs for different processing options. The additional FIDEL facility of the LINE option helps you organize the use of multiple CRTFORMs (see *Using Multiple CRTFORMs: LINE* on page 10-51).
- Case Logic, which enables you to divide the processing into logical subdivisions for particular sets of circumstances (see *Case Logic* in Chapter 9, *Modifying Data Sources With MODIFY*, and *CRTFORMs and Case Logic* on page 10-57).
- Groups of fields (see *Specifying Groups of Fields* on page 10-59).
- VALIDATES and various error handling formats (see *Handling Errors* on page 10-68).
- Log files that preserve a record of all data that is entered onto the screen (see *Logging Transactions* on page 10-73).

MODIFY also has additional screen control options such as clearing the screen, setting the height and width parameters, and changing the default size of the TYPE message area in order to enlarge the CRTFORM (see *Additional Screen Control Options* on page 10-74).

Conditional and Non-Conditional Fields

When you run a MODIFY request, FOCUS keeps track of which transaction fields are active or inactive during execution. In order to add, update, and delete segment instances, the fields must be active (see *Active and Inactive Fields* in Chapter 9, *Modifying Data Sources With MODIFY*, for a full discussion of active and inactive fields).

You can define data entry and turnaround fields as either conditional or non-conditional. A conditional field is conditionally active. That is, it becomes active only if there is incoming data present for the field. Otherwise, it remains inactive. A non-conditional field is always active whether there is incoming data present or not.

When you are performing update operations, there are several important points to keep in mind when you choose whether to specify a field as conditional or non-conditional:

- If data is entered or changed, the data source value is always updated and the field always becomes active. This is true whether the field is conditional or non-conditional.
- If data is not entered or changed, what happens to the data source value is dependent on whether the field is conditional or non-conditional as well as program logic. The following table outlines this.

Type of Field	Active/Inactive	Data Source Value
Conditional Entry	Inactive	Remains. Display value ignored.
Conditional Turnaround	Inactive	Remains. Display value ignored.
Non-Conditional Entry	Active	Displayed value replaces data source value (blank or 0).
Non-Conditional Turnaround	Active	Displayed value replaces data source value (same value).

- If a field is active, the displayed value always becomes the new data source value. In turnaround fields, this is by definition the same value.
- If a field is inactive, the displayed value is always ignored.
- If you compute a data source field and then display it on the CRTFORM with a D. or a T., the field must still be active to get the computed value displayed on the screen. Otherwise, you get a blank or 0.
- When you use a VALIDATE for a field, the field must be active. Otherwise you do not get the accurate data source value validated; instead, you get a blank or 0.

Note: You can make a field active or inactive by using the ACTIVATE or DEACTIVATE statement respectively.

Example Conditional and Non-Conditional Display and Turnaround Fields

The following example illustrates the display and turnaround field features as well as the use of a non-conditional turnaround field (both carets). The first CRTFORM asks for a key field value, in this case EMP_ID. If a matching record is obtained, then some data source values are displayed and others are shown for turnaround update.

Note how the non-conditional turnaround field functions in the following example. Whether the displayed value is changed or not, the value in the data source is active. The VALIDATE uses the display value, whether it was changed or not.

```

MODIFY FILE EMPLOYEE
1. CRTFORM
   "ENTER EMPLOYEE ID#: <EMP_ID"
   "PRESS ENTER BEFORE CONTINUING"
   "-----"
MATCH EMP_ID
  ON NOMATCH TYPE
    "EMPLOYEE ID NOT IN DATABASE. PLEASE REENTER."
  ON NOMATCH REJECT
2. ON MATCH CRTFORM LINE 4
   " "
   "EMPLOYEE ID #: <D.EMP_ID"
   "LAST NAME: <D.LAST_NAME"
   "HIRE DATE: <D.HIRE_DATE"
   "SALARY: <T.CURR_SAL>"
   "DEPARTMENT: <T.DEPARTMENT>"
3. ON MATCH VALIDATE
   SALTEST = IF CURR_SAL GT 0 THEN 1 ELSE 0;
  ON INVALID TYPE
    "SALARY MUST BE GREATER THAN 0"
    "CORRECT SALARY AND PRESS ENTER TWICE"
  ON MATCH UPDATE CURR_SAL DEPARTMENT
DATA
END

```

The example processes as follows:

1. When the procedure executes, the top part of the CRTFORM appears as follows:

```
ENTER EMPLOYEE ID #:  
PRESS ENTER BEFORE CONTINUING  
-----
```

If the employee ID entered does not match an ID in the data source, the transaction is rejected and a TYPE statement appears at the bottom of the screen.

Assume the operator enters the following employee ID:

```
ENTER EMPLOYEE ID #: 818692173  
PRESS ENTER BEFORE CONTINUING  
-----
```

2. If the ID entered matches an ID in the data source, FOCUS successfully retrieves a record. The ON MATCH CRTFORM causes a second CRTFORM to be displayed on line 4. This CRTFORM contains both display and turnaround fields. The data source values of the fields appear on the second CRTFORM, and the cursor is positioned at the start of the CURR_SAL field which is the first unprotected field. Note that both CURR_SAL and DEPARTMENT are automatically highlighted for turnaround:

```
ENTER EMPLOYEE ID #: 818692173  
PRESS ENTER BEFORE CONTINUING  
-----  
  
EMPLOYEE ID #: 818692173  
LAST NAME: CROSS  
HIRE DATE: 811102  
SALARY: 27062.00  
DEPARTMENT: MIS
```

Assume the operator updates DEPARTMENT, does not change CURR_SAL, and transmits the CRTFORM:

```
ENTER EMPLOYEE ID #: 818692173  
PRESS ENTER BEFORE CONTINUING  
-----  
  
EMPLOYEE ID #: 818692173  
LAST NAME: CROSS  
HIRE DATE 811102  
SALARY: 27062.00  
DEPARTMENT: ois
```

- When the operator presses Enter, the transaction is processed. If the value of CURR_SAL is greater than 0, the VALIDATE will evaluate as 1 (true) and processing continues. Although a value was not entered for CURR_SAL, the field is active because it is specified as a non-conditional field. Thus, the VALIDATE reads the current value in the T. field (27062.00), and validates the field. The transaction is then processed.

If you specify the turnaround field as conditional (only the left caret), the field is inactive if no data is entered. Assume the same transaction as above. The operator updates the DEPARTMENT and does not enter new data for the CURR_SAL field. The VALIDATE does not read the T. value because the field is inactive and instead reads a 0. The field is invalidated and the following error message occurs:

```

ENTER EMPLOYEE ID #: 818692173
PRESS ENTER BEFORE CONTINUING
-----

EMPLOYEE ID #:      818692173
LAST NAME:         CROSS
HIRE DATE:         811102
SALARY:            27062.00
DEPARTMENT:        ois

(FOC421)TRANS 1 REJECTED INVALID SALTEST
INVALID SALARY
SALARY MUST BE GREATER THAN 0

```

Using FIXFORM and FIDEL in a Single MODIFY

A MODIFY procedure can mix data sources from CRTFORMs and FIXFORMs.

The rules are:

- You can have only one FIXFORM statement and you must specify the name of the transaction data source. For example:


```
FIXFORM ON filename
```
- The FIXFORM statement must precede the CRTFORM statement.
- START and STOP do not apply (see *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* in Chapter 9, *Modifying Data Sources With MODIFY*).

This feature is useful in situations where a known set of records is wanted and the keys for these records reside on an external fixed format data source. (The data source may have been produced by a prior TABLE and SAVE or HOLD command.) The procedure first reads a key, fetches the matching record, and displays it on the CRTFORM specified.

This is also convenient when the FIXFORM is included in a START case.

In the following example, FIXFORM is used with FIDEL. To run this example on your machine, you must first create a sequential data source with data. To do so, run this TABLE request:

```
TABLE FILE EMPLOYEE
PRINT EMP_ID PAY_DATE
IF PAY_DATE GE 820730
ON TABLE SAVE AS PAYTRANS
END
```

This creates the transaction data source PAYTRANS. Then run the following MODIFY request:

```
MODIFY FILE EMPLOYEE
1. FIXFORM ON PAYTRANS EMP_ID/9 PAY_DATE/6
2. MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH CONTINUE
MATCH PAY_DATE
3. ON MATCH/NOMATCH CRTFORM
   "EMPLOYEE ID #: <D.EMP_ID>"
   "PAY_DATE: <D.PAY_DATE>"
   "MONTHLY GROSS: <T.GROSS>"
   ON NOMATCH INCLUDE
   ON MATCH UPDATE GROSS
DATA
END
```

The example processes as follows:

1. First the data is read in from the sequential data source PAYTRANS.
2. The EMP_ID from PAYTRANS is matched against EMP_IDs in the EMPLOYEE data source. If the EMP_IDs match, PAY_DATE is matched.
3. The CRTFORM shows display values for EMP_ID and PAY_DATE. If there is a match on PAY_DATE, GROSS is displayed as a turnaround field and the operator can update it. If there is no match on PAY_DATE, both PAY_DATE and GROSS are included:

EMPLOYEE ID #:	071382660
PAY_DATE:	820831
MONTHLY GROSS:	916.67

The procedure ends when there are no more transactions to read on the external data source. It can also be terminated by the operator pressing the PF1 or PF3 key.

Generating Automatic CRTFORMs

You can use several simple but powerful statements with the FOCUS MODIFY facility to allow immediate generation of data management requests. You do not need to learn the complete FOCUS MODIFY language. Without using field names, you can write general-purpose requests and customize them for more detailed situations.

The statements can be used with multi-segment data sources as well as simple data sources. They can also be used from the Screen Painter (see *Generating CRTFORMs Automatically* on page 10-96). These statements automatically specify conditional fields. They include:

<code>CRTFORM * [SEG n]</code>	Design screen for all real data fields in segment <i>n</i> , where <i>n</i> is either the segment name or number.
<code>CRTFORM * KEYS [SEG n]</code>	Design screen for all key fields in segment <i>n</i> .
<code>CRTFORM * NONKEYS [SEG n]</code>	Design screen for all non-key fields in segment <i>n</i> .
<code>CRTFORM T.* [SEG n]</code>	Design screen using T.fields in segment <i>n</i>
<code>CRTFORM D.* [SEG n]</code>	Design screen using D.fields in segment <i>n</i> .

Note: The use of CRTFORM * on a COMBINE data source name is illogical and may produce unpredictable results.

Note that you can optionally specify the segment name or number for each of the CRTFORMs. To obtain the segment names and numbers, enter the following command where *file* is the name of the data source:

```
CHECK FILE file PICTURE
```

The names and numbers appear on the top of each segment in the diagram. You may also list segment names and numbers by entering the command:

```
? FDT filename
```

See the *Describing Data* manual and the *Developing Applications* manual for more information on the CHECK FILE command and ? FDT query.

If you are modifying all of the segments in the data source (except for unique segments), you can write the request without using Case Logic. The following example adds and maintains data for the EMPLOYEE data source. The segments are as follows:

- Segment 1 contains basic employee data (names, jobs, salaries, and so on).
- Segment 3 contains employee salary histories.
- Segment 7 stores employees' home addresses and information on their bank accounts.
- Segment 8 stores employee monthly pay.
- Segment 9 stores monthly deductions.

(Segment 2 is a unique segment. Segments 4, 5, and 6 are cross-referenced segments that are not part of the EMPLOYEE data source.)

The request is:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "THIS PROCEDURE ADDS NEW RECORDS AND UPDATES EXISTING RECORDS </1"
  "INSTRUCTIONS"
  "1. ENTER DATA FOR EACH FIELD"
  "2. USE TAB KEY TO MOVE CURSOR"
  "3. PRESS ENTER WHEN FINISHED"
  "4. WHEN YOU FINISH ALL RECORDS, PRESS PF1 </1"
CRTFORM * KEYS
MATCH * KEYS SEG 01
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 01
  ON MATCH UPDATE * SEG 01
  ON NOMATCH INCLUDE
MATCH * KEYS SEG 03
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 03
  ON MATCH UPDATE * SEG 03
  ON NOMATCH INCLUDE
MATCH * KEYS SEG 07
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 07
  ON MATCH UPDATE * SEG 07
  ON NOMATCH INCLUDE
MATCH * KEYS SEG 08
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 08
  ON MATCH UPDATE * SEG 08
  ON NOMATCH INCLUDE
MATCH * KEYS SEG 09
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 09
  ON MATCH UPDATE * SEG 09
  ON NOMATCH INCLUDE
DATA
END
```

When the procedure executes, the screen appears as follows:

```

THIS PROCEDURE ADDS NEW RECORDS AND UPDATES EXISTING RECORDS

INSTRUCTIONS
1. ENTER DATA FOR EACH FIELD
2. USE TAB KEY TO MOVE CURSOR
3. PRESS ENTER WHEN FINISHED
4. WHEN YOU FINISH ALL RECORDS, PRESS PF1

EMP_ID:      :
DAT_INC:     :
TYPE:        :
PAY_DATE:    :
DED_CODE:    :

LAST_NAME:   :                FIRST_NAME:      :
HIRE_DATE:   :                DEPARTMENT:    :
CURR_SAL:    :                CURR_JOBCODE:  :
ED_HRS:      :

PCT_INC:     :                SALARY:         :
JOBCODE:     :

ADDRESS_LN1: :
ADDRESS_LN2: :
ADDRESS_LN3: :

ACCTNUMBER:  :
GROSS:       :

```

Notice that the fields are divided into five groups. The first group consists of all the key fields in the data source. Each subsequent group consists of all non-key fields in a particular segment. Fill in each group from top to bottom and press Enter before filling in the next group. When you do this, the request uses the values to match on the segments specified later in the request.

The first CRTFORM statement generates the first group of fields, which are all the key fields in the data source:

```
CRTFORM * KEYS
```

The MATCH statements in the request modify each of the segments in the data source. Each statement contains a CRTFORM phrase that prompts for all non-key fields in the segment:

```
CRTFORM T.* NONKEYS SEG xx
```

Note that the CRTFORM phrase displays the fields as turnaround fields. After you fill in the fields in the group and press Enter, FOCUS uses the field values to update the segment.

You can add the following enhancements to the request:

- The LINE option on each CRTFORM statement.
- Explanatory text after each CRTFORM statement.
- A separate CRTFORM containing explanatory text at the beginning of the request.

If you want to modify some but not all segments in the data source, use Case Logic to write the request. Place each MATCH statement in a separate case. For example, this request modifies data in Segments 1, 3, and 7:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "THIS PROCEDURE MAINTAINS EMPLOYEE"
  "JOB DATA, SALARY HISTORIES, AND ADDRESSES"
  " "
CRTFORM * KEYS
  "FILL IN EMP_ID, DAT_INC, AND TYPE FIELDS"
  "THEN PRESS ENTER"
GOTO EMPLOYEE

CASE EMPLOYEE
MATCH * KEYS SEG 01
  ON NOMATCH REJECT
  ON MATCH CRTFORM T.* NONKEYS SEG 01 LINE 10
  ON MATCH UPDATE * SEG 01
  ON MATCH GOTO MONTHPAY
ENDCASE

CASE MONTHPAY
MATCH * KEYS SEG 03
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 03 LINE 10
  ON MATCH UPDATE * SEG 03
  ON MATCH GOTO DEDUCT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO DEDUCT
ENDCASE

CASE DEDUCT
MATCH * KEYS SEG 07
  ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 07 LINE 10
  ON MATCH UPDATE * SEG 07
  ON NOMATCH INCLUDE
ENDCASE
DATA
END
```


Using Multiple CRTFORMs: LINE

You can choose which screen line the CRTFORM will begin on by using the LINE option. By default, the first CRTFORM begins on line 1. The next CRTFORM in the procedure begins on the line following the end of the previous CRTFORM. For example, if one screen uses 12 lines, the next CRTFORM automatically begins on the 13th line.

In the following example, there are two logical forms: EMPLOYEE UPDATE and FUND TRANSFER INFORMATION UPDATE. It illustrates the placement of CRTFORMs when the default is in effect (that is, the LINE option is not used):

```

MODIFY FILE EMPLOYEE
1. CRTFORM
    "EMPLOYEE UPDATE"
    " "
    "-----"
    "EMPLOYEE ID #: <EMP_ID      LAST_NAME: <LAST_NAME"
    "
    "DEPARTMENT:      <DEPARTMENT <28 SALARY: <CURR_SAL"
    " "
    "BANK: <BANK_NAME"
    " "
    "FILL IN THE ABOVE FORM AND PRESS ENTER"
    "-----"
MATCH EMP_ID
ON NOMATCH REJECT
ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
ON MATCH CONTINUE TO BANK_NAME
    ON NOMATCH INCLUDE
2.    ON MATCH/NOMATCH CRTFORM
    "</1"
    "FUND TRANSFER INFORMATION UPDATE"
    " "
    "-----"
    "BANK: <D.BN  ACCT #: <T.BA"
    " "
    "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
    "-----"
        ON MATCH UPDATE BA BC EDATE
DATA
END

```

This produces the following screen when the request is executed:

```
EMPLOYEE UPDATE
-----
EMPLOYEE ID #:          LAST_NAME:
DEPARTMENT:            SALARY:
BANK:
FILL IN THE ABOVE FORM AND PRESS ENTER
-----
FUND TRANSFER INFORMATION UPDATE
-----
BANK:                  ACCT #:
BANK CODE:             START DATE:
-----
```

Note that when the default is in effect, each logical form is displayed one after the other on the screen, the instant the MODIFY procedure is executed. That is, all the distinct CRTFORMs are concatenated into one visual form.

The LINE option enables you to control both the placement of a CRTFORM on the screen and the timing with which it appears on the screen. Using LINE gives you the following options:

- You can have one logical form replace another after each transaction by having subsequent CRTFORMs begin on the same line.
- You can build mixed screens by saving lines from a previous CRTFORM on the screen while executing a subsequent CRTFORM. In other words, the first CRTFORM is displayed, the operator transmits the data, and the next CRTFORM is displayed while the previous one remains on the screen.

The syntax is

```
CRTFORM [LINE nn]
```

where:

nn

Is the starting line number for the CRTFORM.

To completely replace one screen with the next, both CRTFORMs must start on the same line. Note the following change in the previous example:

```
MODIFY FILE EMPLOYEE
1. CRTFORM
   "EMPLOYEE UPDATE"
   " "
   "-----"
   "EMPLOYEE ID #:  <EMP_ID LAST_NAME:  <LAST_NAME"
   " "
   "DEPARTMENT:      <DEPARTMENT <30 SALARY:  <CURR_SAL"
   " "
   "BANK:             <BANK_NAME"
   " "
   "FILL IN THE ABOVE FORM AND PRESS ENTER"
   "-----"
MATCH EMP_ID
ON NOMATCH REJECT
ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
ON MATCH CONTINUE TO BANK_NAME
ON NOMATCH INCLUDE
```

```
2.  ON MATCH/NOMATCH CRTFORM LINE 1
    "</1"
    "FUND TRANSFER INFORMATION UPDATE"
    " "
    "-----"
    "BANK: <D.BN ACCT #: <T.BA"
    " "
    "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
    "-----"
    ON MATCH UPDATE BA BC EDATE
DATA
END
```

1. When the MODIFY procedure is executed, the following screen is displayed:

```
EMPLOYEE UPDATE
-----
EMPLOYEE ID #:          LAST_NAME:
DEPARTMENT:            SALARY:
BANK:
FILL IN THE ABOVE FORM AND PRESS ENTER
-----
```

2. After the operator enters and transmits the data, the next CRTFORM replaces the previous one on the screen:

```
FUND TRANSFER INFORMATION UPDATE
-----
BANK:                  ACCT #:
BANK CODE:            START DATE:
-----
```

Generally, it is a good practice to put LINE 1 on all CRTFORMs that start a new case (see *CRTFORMs and Case Logic* on page 10-57) unless a specific screen pattern is wanted.

A combination of two or more individual CRTFORMs can occupy specific lines on one screen. To obtain a mixed screen, place the desired starting line number with the CRTFORM statement. For instance:

```

MODIFY FILE EMPLOYEE
1. CRTFORM
    "EMPLOYEE UPDATE"
    " "
    "-----"
    "EMPLOYEE ID #:    <EMP_ID LAST_NAME: <LAST_NAME"
    " "
    "DEPARTMENT:      <DEPARTMENT <30 SALARY: <CURR_SAL"
    " "
    "BANK:             <BANK_NAME"
    " "
    "FILL IN THE ABOVE FORM AND PRESS ENTER"
    "-----"

MATCH EMP_ID
ON NOMATCH REJECT
ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
ON MATCH CONTINUE TO BANK_NAME
ON NOMATCH INCLUDE
2. ON MATCH/NOMATCH CRTFORM LINE 12
    "</1"
    "FUND TRANSFER INFORMATION UPDATE"
    " "
    "-----"
    "BANK: <D.BN ACCT #: <T.BA"
    " "
    "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
    "-----"
ON MATCH UPDATE BA BC EDATE

DATA
END

```

Processing occurs as follows:

1. Like the preceding examples, this produces the first screen. Assume the operator enters and transmits the following data:

```
EMPLOYEE UPDATE
-----
EMPLOYEE ID #:  117593129          LAST_NAME:  JONES
DEPARTMENT:    MIS                SALARY:     18480
BANK:          STATE
FILL IN THE ABOVE FORM AND PRESS ENTER
-----
```

2. The first CRTFORM remains on the screen while the next CRTFORM is displayed on line 12:

```
EMPLOYEE UPDATE
-----
EMPLOYEE ID #:  117593129          LAST_NAME:  JONES
DEPARTMENT:    MIS                CURRENT SALARY:  18480
BANK:          STATE
FILL IN THE ABOVE FORM AND PRESS ENTER
-----
FUND TRANSFER INFORMATION UPDATE
-----
BANK: STATE                ACCT #:40950036
BANK CODE: 510271         START DATE:821101
-----
```

You can save certain lines from the preceding CRTFORM while you discard others. In the previous example, if you begin the second CRTFORM on line 6, the EMP_ID and the LAST_NAME remain and the next line is the beginning of the FUND TRANSFER INFORMATION AND UPDATE.

Assume the operator enters and transmits data on the first CRTFORM. Part of the first logical form disappears and the second form is displayed. Thus, a new visual form is created:

```

EMPLOYEE UPDATE
-----
EMPLOYEE ID #:    117593129          LAST_NAME:    JONES
FUND TRANSFER INFORMATION AND UPDATE
-----
BANK:    STATE          ACCT #:    40950036
BANK CODE:  510271          START DATE:   821101
-----

```

You can create mixed screens using the LINE option, in a variety of ways, depending on the need of the application.

CRTFORMs and Case Logic

Case Logic, described in *Case Logic* in Chapter 9, *Modifying Data Sources With MODIFY*, enables you to perform separate complete MODIFY processes in one procedure. Each of these is a case, and the procedure contains directions about which case to execute under various circumstances.

When you use the Case Logic option of the MODIFY command, you can create a pattern of many CRTFORMs.

When there are multiple CRTFORMs in a single MODIFY request, use the LINE option to specify where each CRTFORM will be displayed. With Case Logic, generally, we recommend that you use LINE 1 to replace one screen with another.

The following example illustrates the use of Case Logic with the CRTFORM:

```
MODIFY FILE EMPLOYEE
COMPUTE
  PFKEY/A4= ;
CRTFORM
  "TO INPUT A NEW RECORD, PRESS PF4"
  "TO UPDATE AN EXISTING RECORD, PRESS PF5"
IF PFKEY EQ 'PF04' GOTO ADD ELSE
IF PFKEY EQ 'PF05' GOTO UPP ELSE GOTO TOP;

CASE ADD
CRTFORM LINE 1
  "EMPLOYEE ID #:      <EMP_ID"
  "LAST NAME:         <LAST_NAME FIRST NAME: <FIRST_NAME"
  "HIRE DATE:         <HIRE_DATE"
  "DEPARTMENT:        <DEPARTMENT"
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
ENDCASE

CASE UPP
CRTFORM LINE 1
  "EMPLOYEE ID #:      <EMP_ID"
  "DEPARTMENT:         <DEPARTMENT"
  "JOB CODE:           <CURR_JOBCODE"
  "SALARY:             <CURR_SAL"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_JOBCODE CURR_SAL
ENDCASE
DATA
END
```

The first CRTFORM appears as:

```
TO INPUT A NEW RECORD, PRESS PF4
TO UPDATE AN EXISTING RECORD, PRESS PF5
```

If the operator presses PF4, the following is displayed:

```
EMPLOYEE ID #:
LAST NAME:           FIRST NAME:
HIRE DATE:
DEPARTMENT:
```


If the operator presses PF5, the following is displayed:

```
EMPLOYEE ID #:
DEPARTMENT:
JOB CODE:
SALARY:
```

Note: At the end of a MODIFY procedure, control defaults to the TOP Case.

Specifying Groups of Fields

Groups of fields (that is, more than one occurrence of the same field) can be specified on the CRTFORM in two ways:

- Specifying a field more than once on a CRTFORM.
- Using REPEAT syntax.

You can use Case Logic to process groups of fields.

Specifying Groups of Fields for Input

A group of fields may repeat on the form. For example:

```
"EMPLOYEE ID  DEPARTMENT  SALARY"
"<EMP_ID      <DPT        <CURR_SAL"
"<EMP_ID      <DPT        <CURR_SAL"
"<EMP_ID      <DPT        <CURR_SAL"
```

This reads the same data as the FIXFORM statement:

```
FIXFORM 3 (EMP_ID/C9 DPT/C10 CURR_SAL/C14)
```

The following example shows the use of repeating groups for a single employee ID:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID #: <EMP_ID"
  " "
  "ENTER PAY DATE AND GROSS PAY FOR ABOVE EMPLOYEE"
  " "
  "PAY DATE: <PAY_DATE      GROSS: <GROSS"
  "PAY DATE: <PAY_DATE      GROSS: <GROSS"
  "PAY DATE: <PAY_DATE      GROSS: <GROSS"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
END
```

Note: A group of repeated data fields cannot be specified on a MATCH or NOMATCH CRTFORM. They must be presented on a primary CRTFORM (that is, one not generated as a result of a MATCH or NOMATCH command).

This procedure processes as follows:

```
ENTER EMPLOYEE ID #:  818692173

ENTER PAY DATE AND GROSS AMOUNT FOR ABOVE EMPLOYEE

PAY DATE: 850405      GROSS: 3000.00
PAY DATE: 850412      GROSS: 4000.00
PAY DATE: 850418      GROSS: 2500.00
```

When the operator presses Enter, the transaction processes. Processing continues until a line with no data is found or all lines are completed (whichever occurs first).

Using REPEAT to Display Multiple Records

You can display multiple segment instances on the screen by directing FIDEL to read and display the contents of a HOLD buffer. You can use a subscript value to identify a particular instance in the HOLD buffer with the following syntax

field(n)

where:

field

Is the name of a previously held field.

(n)

Is the integer subscript that identifies the number of the instance in the HOLD buffer containing the field to be displayed. *n* must be in integer format or the report group will be ignored.

The variable SCREENINDEX allows you to display and modify selected groups of records from the HOLD buffer.

Consider the following example, which uses the REPEAT statement to retrieve up to a set number (in this case, six) of multiple instances, saves them in the HOLD buffer, and then displays the instances on the CRTFORM:

```

MODIFY FILE EMPLOYEE
1. CRTFORM
   "PAY HISTORY UPDATE"
   " "
   "ENTER EMPLOYEE ID#: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO COLLECT

CASE COLLECT
2. REPEAT 6 TIMES
2.   NEXT PAY_DATE
2.   ON NEXT HOLD PAY_DATE GROSS
3.   ON NONEXT GOTO DISPLAY

```

Using FIDEL in MODIFY

```
3.  ENDREPEAT
    GOTO DISPLAY
    ENDCASE

    CASE DISPLAY
    IF HOLDCOUNT EQ 0 GOTO TOP;
4.  COMPUTE
    BUFFNUMBER/I5 = HOLDCOUNT;
5.  CRTFORM LINE 5
    "FILL IN GROSS AMOUNT FOR EACH PAY DATE"
    " "
    "PAY DATE          GROSS AMOUNT"
    "-----          -"
    "<D.PAY_DATE(1)    <T.GROSS(1)>"
    "<D.PAY_DATE(2)    <T.GROSS(2)>"
    "<D.PAY_DATE(3)    <T.GROSS(3)>"
    "<D.PAY_DATE(4)    <T.GROSS(4)>"
    "<D.PAY_DATE(5)    <T.GROSS(5)>"
    "<D.PAY_DATE(6)    <T.GROSS(6)>"
    GOTO UPDATE
    ENDCASE

    CASE UPDATE
6.  REPEAT BUFFNUMBER
    MATCH PAY_DATE
    ON NOMATCH REJECT
    ON MATCH UPDATE GROSS
    ENDREPEAT
    GOTO COLLECT
    ENDCASE
DATA
END
```

The procedure processes as follows:

1. When the procedure is executed, the first CRTFORM is displayed:

```
PAY HISTORY UPDATE
ENTER EMPLOYEE ID #:
```

2. Assume the operator enters the following ID and transmits the data:

```
ENTER EMPLOYEE ID #: 071382660
```

If there is a match, the instruction is to REPEAT the logic six times. That is, up until six times, find a PAY_DATE and hold the PAY_DATE and the GROSS in the HOLD buffer.

3. When there are no more PAY_DATE fields or six of them have been held, the procedure branches to CASE DISPLAY.
4. The procedure stores the number of records that are in the HOLD buffer in the variable BUFFNUMBER.
5. The procedure displays the following CRTFORM:

```
PAY HISTORY UPDATE
ENTER EMPLOYEE ID #: 071382660
FILL IN GROSS AMOUNT FOR EACH PAY DATE

PAY DATE          GROSS AMOUNT
820831             916.67
820730             916.67
820630             916.67
820528             916.67
820430             916.67
820331             916.67
```

The operator makes changes to the fields in the GROSS AMOUNT column and presses Enter. All changes for all records are transmitted simultaneously as shown:

```
PAY HISTORY UPDATE
ENTER EMPLOYEE ID #: 071382660
FILL IN GROSS AMOUNT FOR EACH PAY DATE
PAY DATE          GROSS AMOUNT
820831            816.67
820730            816.67
820630            816.67
820528            916.67
820430            916.67
820331            916.67
```

6. The REPEAT statement instructs FOCUS to perform the MODIFY logic on all segment instances.

Note: If a CRTFORM screen with subscripted variables is rejected with a FORMAT ERROR, you may not alter any records on the screen prior to the record rejected, as FOCUS has already held them.

Using Groups of Fields With Case Logic

When you use Case Logic to process a group of fields, some important rules apply:

- Each time the procedure enters the case, the next group of fields is processed. FOCUS keeps track internally of which groups have been processed.
- If the CRTFORM with the group of fields is not in the TOP case, you must create your own branching logic to process all the groups before going back to the TOP. This normally requires some kind of counting mechanism. Once the procedure goes back to the TOP case, all unprocessed data on the CRTFORM or in a lowercase is lost.

Example Case Logic, Groups, CURSORINDEX and VALIDATE

In the following example, Case Logic is used with groups of fields. The CURSORINDEX (see *Specifying Cursor Position* on page 10-33) is used in conjunction with a VALIDATE:

```

MODIFY FILE EMPLOYEE
1. CRTFORM
   "EMPLOYEE SALARY AND DEPARTMENT UPDATE"
   " "
   "PRESS ENTER"
GOTO COLLECT

CASE COLLECT
2. REPEAT 6 TIMES
   NEXT EMP_ID
   ON NEXT HOLD EMP_ID CURR_SAL DEPARTMENT
   ON NONEXT GOTO DISPLAY
ENDREPEAT
GOTO DISPLAY
ENDCASE

CASE DISPLAY
3. IF HOLDCOUNT EQ 0 GOTO EXIT;
4. COMPUTE
   BUFFNUMBER/I5 = HOLDCOUNT;
5. CRTFORM LINE 7
   "EMPLOYEE           SALARY           DEPARTMENT"
   "-----           -"
   "<D.EMP_ID(1)       <:AAA.T.CSAL(1) >   <:BBB.T.DPT(1) >"
   "<D.EMP_ID(2)       <:AAA.T.CSAL(2) >   <:BBB.T.DPT(2) >"
   "<D.EMP_ID(3)       <:AAA.T.CSAL(3) >   <:BBB.T.DPT(3) >"
   "<D.EMP_ID(4)       <:AAA.T.CSAL(4) >   <:BBB.T.DPT(4) >"
   "<D.EMP_ID(5)       <:AAA.T.CSAL(5) >   <:BBB.T.DPT(5) >"
   "<D.EMP_ID(6)       <:AAA.T.CSAL(6) >   <:BBB.T.DPT(6) >"

```

```
6. REPEAT 6 TIMES
    COMPUTE
        CURSOR/A66 = ':AAA';
        CURSORINDEX/I5=REPEATCOUNT;
    VALIDATE
        SALTEST = IF CSAL GT 50000 THEN 0 ELSE 1;
        ON INVALID TYPE "SALARY MUST BE LESS THAN $50,000"
        ON INVALID GOTO DISPLAY
    ENDREPEAT
    GOTO UPDATE
    ENDCASE

CASE UPDATE
7. REPEAT BUFFNUMBER
    MATCH EMP_ID
        ON NOMATCH REJECT
        ON MATCH UPDATE CURR_SAL DEPARTMENT
    ENDREPEAT
    GOTO COLLECT
    ENDCASE
DATA
END
```

The example processes as follows:

1. The first CRTFORM requests the operator to press Enter without typing anything.
2. The REPEAT statement retrieves six employee IDs, salaries, and department assignments and places them in a buffer.
3. If there are no records in the buffer, the procedure terminates.
4. The COMPUTE statement stores the number of records in the buffer in the variable BUFFNUMBER.
5. The second CRTFORM retrieves the IDs, salaries, and department assignments from the buffer and displays them together on the screen. Note the field labels:
 - The label :AAA on the CURR_SAL (CSAL) field.
 - The label :BBB on the DEPARTMENT (DPT) field.

Assume that the operator changes the values to the following:


```

EMPLOYEE SALARY AND DEPARTMENT UPDATE

PRESS ENTER

EMPLOYEE          SALARY          DEPARTMENT
-----          -
071382660        35000.00        PRODUCTION
112847612        23200.00        MIS
117593129        75480.00        MIS
119265415        19500.00        PRODUCTION
119329144        39700.00        PRODUCTION
123764317        36862.00        PRODUCTION
    
```

6. The second REPEAT statement operates on each of the six records displayed by the second CRTFORM, in order of display, performing the following tasks:
 - Sets the CURSOR variable to the label :AAA.
 - Sets the CURSORINDEX variable to the number of the record it's processing (1 through 6).
 - Validates the CURR_SAL field value. If the CURR_SAL value is \$50,000 or more, the procedure branches back to the beginning of Case DISPLAY. The procedure displays the second CRTFORM again, with the CURSOR and CURSORINDEX variables positioning the cursor on the invalid salary.

In the example, the procedure positions the cursor on the third CURR_SAL value:

```

EMPLOYEE SALARY AND DEPARTMENT UPDATE

PRESS ENTER

EMPLOYEE          SALARY          DEPARTMENT
-----          -
071382660        35000.00        PRODUCTION
112847612        23200.00        MIS
117593129        _75480.00        MIS
119265415        19500.00        PRODUCTION
119329144        39700.00        PRODUCTION
123764317        36862.00        PRODUCTION

(FOC421)TRANS 2 REJECTED INVALID SALTEST
SALARY MUST BE LESS THAN $50,000
    
```

7. If all values are valid, the third REPEAT statement updates the employee's salary and department for each record in the buffer. The procedure then branches to Case COLLECT to update six more records in the data source.

Handling Errors

It is important to know how various errors are handled by FIDEL so that proper instructions can be given to terminal operators. The following errors can cause a transaction or screen of data to be rejected:

- A format error, caused by entering non-numeric data for a numeric field.
- A validation error, caused by entering an incoming value that failed a VALIDATE test coded in the MODIFY.
- A NOMATCH condition, caused by entering data for a key field that did not match any record in the data source.
- A DUPLICATE condition, caused by key field values that matched records on a data source.
- An ACCEPT error, caused by entering a value for a data source field that failed the ACCEPT test.

Note: Error messages are discussed in detail in *Messages: TYPE, LOG, and HELPMESSAGE* in Chapter 9, *Modifying Data Sources With MODIFY*.

Handling Format Errors

If the operator enters a non-numeric character into a field defined as numeric, an error message is displayed and the screen is not processed (processing stops). The error message indicates the line number and field name in error and the cursor is automatically positioned on that field. Additionally, if the operator enters a value that fails an ACCEPT test for a field an error message is displayed and the screen is not processed. Any message specified for that field with the HELPMESSAGE attribute will also be displayed.

The operator can retype the data and press the Enter key to retransmit the screen. Alternatively, the operator may press the PF2 key to cancel the transaction. The error prevents anything on the screen from being processed. When the operator corrects the error and transmits the screen, processing resumes.

There are two exceptions to this rule. When there are repeating groups, all complete transactions up to the error will be processed. Also, in REPEAT/HOLD loops, the data prior to the format error may not be altered.

VALIDATE and CRTFORM Display Logic

When the operator enters a value that is invalid, the transaction is rejected and an error message is displayed. By default, control returns to the first CRTFORM in the TOP case. However, you can use an ON INVALID GOTO statement to transfer control to any other case in the request.

If the NOCLEAR or blank option in the CRTFORM statement (see *Additional Screen Control Options* on page 10-74) is in effect, the screen will not be cleared. The operator can change the data in the offending transaction and retransmit the screen.

When you use validations, you can divide the tests into different cases and repeat a case if it fails the test. The advantage of this is that the operator can change the invalid data and retransmit the screen before other sections are processed. An ON INVALID TYPE phrase can be used to send an informative message to the operator on the screen. The following example shows the use of these options:

```
CASE TRY
CRTFORM
    EMPLOYEE ID #: <EMP_ID NAME: <LAST_NAME"
    "CURRENT SALARY: <CURR_SAL"
VALIDATE
    GOODSAL= CURR_SAL GT 10000 AND CURR_SAL LT 1000000;
    ON INVALID TYPE
    THE CURRENT SALARY CANNOT BE LARGER THAN 1000000 OR"
    "LESS THAN 10000"
    ON INVALID GOTO TRY
.
.
.
```

All messages appear on the bottom four lines of the screen, unless you specify the TYPE option on the CRTFORM statement (see *Additional Screen Control Options* on page 10-74).

Handling Errors With Repeating Groups

If old style repeating groups (those without subscripts) are present and there is an error, processing continues to the next transaction on the screen. This means that if the operator changes the offending transaction and retransmits the screen, the other transactions on the screen become duplicates. It is important when using repeating groups to reject duplicates and turn the duplicate message off (LOG DUPL MSG OFF).

Alternatively, avoid using VALIDATE with repeating groups. Use COMPUTE instead and branch to a case that displays the erroneous data in a lower portion of the screen.

The following is an example of this technique. A test field is computed in Case TEST, using DECODE. This test field checks that the department value is a valid one. If the operator inputs a department value that is invalid, control branches to a case that displays the erroneous data (CASE BADDPT).

```
MODIFY FILE EMPLOYEE
1. CRTFORM
   "FILL IN THE FOLLOWING CHART WITH THE SALARIES"
   "AND DEPARTMENT ASSIGNMENTS OF FOUR NEW EMPLOYEES"
   " "
   "          EMPLOYEE ID          DEPARTMENT          SALARY"
   "          -----          -"
   "PERSON 1  <EMP_ID              <DEPARTMENT        <CURR_SAL"
   "PERSON 2  <EMP_ID              <DEPARTMENT        <CURR_SAL"
   "PERSON 3  <EMP_ID              <DEPARTMENT        <CURR_SAL"
   "PERSON 4  <EMP_ID              <DEPARTMENT        <CURR_SAL"
   GOTTO TEST

2. CASE TEST
   IF EMP_ID IS ' ' GOTO TOP;
   COMPUTE
     TEST/I1 = DECODE DEPARTMENT (MIS 1 PRODUCTION 1 ELSE 0);
   IF TEST IS 0 GOTO BADDEPT ELSE GOTO ADD;
   ENDCASE

3. CASE ADD
   MATCH EMP_ID
     ON NOMATCH INCLUDE
     ON MATCH REJECT
   ENDCASE
```

```

4. CASE BADDEPT
  COMPUTE
    XEMP/A9 = EMP_ID;
    XDEPT/A10 = DEPARTMENT;
  CRTFORM LINE 12
    "INVALID ENTRY: DEPARTMENT MUST BE MIS OR PRODUCTION"
    "CORRECT THE ENTRY BELOW"
    " "
    "EMPLOYEE ID: <D.XEMP   DEPARTMENT: <T.XDEPT"
  COMPUTE
    DEPARTMENT=XDEPT;
  GOTO TEST
  ENDCASE

  DATA
  END

```

The request processes as follows:

1. This is the first and TOP case, and contains a CRTFORM that displays four instances of repeating groups. Assume the operator fills in values and transmits the screen. Control transfers to Case TEST.
2. Case TEST contains a computed field that uses DECODE to make sure that the values that have been input for DEPARTMENT are either MIS or PRODUCTION. When a DEPARTMENT value does not match this list, TEST is returned a code of 0, in which case control transfers to Case BADDEPT.
3. Case BADDEPT first computes two fields, XEMP and XDEPT, to have the values of EMP_ID and DEPARTMENT at the time the error occurred. Next, BADDEPT displays a CRTFORM containing a message to the operator and the two computed fields. The XDEPT field, which contains the invalid DEPARTMENT value, is a turnaround field so that the operator can see the invalid value and change it. Next, the COMPUTE is reversed and the new values are returned to their respective fields. Control transfers back to Case TEST where the DEPARTMENT values will continue to be tested until they are all valid. At that point, control transfers to Case ADD.
4. Case ADD contains the MATCH logic necessary to include new employees into the EMPLOYEE data source. The transaction including all the repeating groups is processed at one time.

Rejecting NOMATCH or Duplicate Data

When the request directs that transactions be rejected, an error message is displayed on the screen. It is a good idea, when the major keys do not repeat, to use a split CRTFORM and give the keys in the first CRTFORM. Once the keys are accepted, the rest of the data may be entered. For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID#:      <EMP_ID"
  "THEN PRESS ENTER"
MATCH EMP_ID
  ON NOMATCH TYPE
    "ID NOT IN DATABASE   PLEASE REENTER"
  ON NOMATCH REJECT
  ON MATCH CRTFORM LINE 4
    "LAST NAME:      <T.LAST_NAME"
    "DEPARTMENT:    <T.DEPARTMENT"
    "SALARY:        <T.CURR_SAL"
  ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
DATA
END
```

If the EMP_ID does not match, control returns immediately to the operator with a request to correct the value. If a match does occur, the operator must then fill in the balance of the form and transmit it.

If repeating groups are present and no other cases are involved, all of the groups are processed before control returns to the screen. Thus, splitting screens in this way is particularly useful when the second CRTFORM contains repeating groups.

Logging Transactions

You can log the data entered on the screen to any log file. Only the data is logged, not the CRTFORM, so a compact log file is created. For example:

```
LOG TRANS ON ALLDATA
```

This will log transactions to a file allocated to the ddname ALLDATA.

The record length of the file must allow space for each field on each CRTFORM in the procedure, plus one character at the start of each CRTFORM. The record length should not be longer than this.

This may be an inconvenient format, since it is very long if several CRTFORMs exist. Instead you can construct a custom log file of your own design using TYPE statements. This example logs data collected from its preceding CRTFORM to a file allocated to ddname MYCRT, including a COMPUTE transaction number, TNUM:

```
CRTFORM
"EMPLOYEE ID #: <EMP_ID  NAME <LAST_NAME"
"HIRE DATE: <HIRE_DATE"
COMPUTE
TNUM/I4=TNUM+1;
TYPE ON MYCRT
"<TNUM><EMP_ID><LAST_NAME><HIRE_DATE"
```

This option is preferable to the standard LOG option whenever a procedure contains more than two CRTFORMs, or when text or computed fields must be captured on the log file.

Additional Screen Control Options

MODIFY CRTFORMs support several additional screen control options:

- Clearing the screen with CLEAR/NOCLEAR.
- Specifying the screen size with WIDTH/HEIGHT.
- Changing the size of the message area at the bottom of the screen using TYPE. This increases the length of the screen that can be used for the actual form.

Clearing the Screen: CLEAR/NOCLEAR

Data is transmitted from the CRTFORM to the data source when the operator presses the Enter key. After each successful screen is processed, the data areas are automatically cleared. You can override this default by using the NOCLEAR option. Then, after each data transmission, the screen remains unchanged.

This is a useful feature when there is a substantial amount of data that carries over from one screen to another. The syntax is

`CRTFORM action`

where:

`action`

Is one of the following:

`blank` is the default. Causes the screen to clear after the data is transmitted. If a transaction is invalid and an error message appears at the bottom of the screen, the screen will not be cleared.

`NOCLEAR` causes the data values on the screen to remain as is after data is transmitted.

`CLEAR` causes the data values on the screen to clear after every data transmission, even if there is an error. Thus, if CLEAR is specifically used and there is an error, data must be reentered.

Note: The options chosen may be different from one CRTFORM to the next.

Specifying Screen Size: WIDTH/HEIGHT

FIDEL assumes a default screen size of 24 lines of 80 characters each. The WIDTH/HEIGHT options allow you to use the full width and height of IBM terminals that are larger than the usual 3270 screen for the display of CRTFORMs. The following syntax allows you to override the defaults

```
CRTFORM [WIDTH nnn] [HEIGHT nnn]
```

where:

WIDTH *nnn*

Is the total number of characters across the face of the screen. Acceptable values for WIDTH are 80 and 132 and cannot exceed the true width of the terminal. FOCUS verifies that each line of the CRTFORM can be displayed at the current WIDTH specification. If any line of the CRTFORM exceeds it, you will receive error message FOC456, and the procedure will not run.

HEIGHT *nnn*

Is the total number of lines that each screen supports. It bears no relation to the number of lines in the CRTFORM. It may not exceed the true height of the terminal, but it may be less. For example, you can specify HEIGHT 20 for a Model 2 screen instead of 24 and write a CRTFORM of 32 lines. The first 16 lines appear on one screen and the next 16 on the subsequent screen. Remember that by default, four lines are reserved for TYPE messages.

The following table gives the physical screen sizes for the IBM 3270 series of terminals:

Terminal Type	Model	Width	Height
3270	1	80	24
3277, 3278, 3279, 3178	2	80	24
3278, 3279	3	80	32
3278	4	80	43
3278	5	132	27

FOCUS senses the width and height of the terminal which you are using and attempts to implement your CRTFORM WIDTH and HEIGHT specifications accordingly. Here are some rules and facts that apply:

- If your WIDTH or HEIGHT specifications exceed the perceived characteristics of the terminal, you will receive a FOC491 error message and the procedure will not run.
- FOCUS sees the terminal as it is defined to the operating system. For example, a Model 5 3278 may be defined to the operating system as a Model 2 terminal. That terminal will appear to FOCUS as a Model 2 (24 lines deep and 80 characters wide). A WIDTH 132 specification will produce a FOC491 error message.
- The following special considerations apply to the Model 5 terminal, which can display 80 columns of normal sized characters, or 132 columns of reduced characters (this choice of character sets is only available under CMS; under MVS, you can only get the reduced character set):

Due to hardware characteristics, you will always get the reduced character set if you use more than 24 lines of the screen. In other words, if you specify a HEIGHT greater than 24, or if you say HEIGHT *, which means 27 lines on the Model 5, you will get the reduced character set, regardless of the WIDTH setting.

To get the normal sized character set, do not specify HEIGHT or WIDTH. You will get a 24x80 screen as if you were on a Model 2 terminal. However, if you specify WIDTH explicitly, FOCUS will automatically use the reduced character set, regardless of the HEIGHT setting.

Changing the Size of the Message Area: TYPE

By default, FOCUS reserves the last four lines of the terminal screen for TYPE messages and text messages specified with the HELPMESSAGE attribute (see *Messages: TYPE, LOG, and HELPMESSAGE* in Chapter 9, *Modifying Data Sources With MODIFY*). You can override this default and determine the number of lines each CRTFORM reserves with the keyword TYPE. This feature allows you to increase the number of lines on the screen for CRTFORM display and reduce the number of lines reserved for messages at the bottom of the screen. The syntax is

```
CRTFORM TYPE {n|4}
```

where:

n

Is a number from one to four indicating the number of message lines desired. The TYPE value setting remains in effect for all subsequent CRTFORMs in the same procedure until overridden by a new value.

You can expand the actual CRTFORM screen size by specifying a number less than four. For example, a terminal with a height of 24 lines currently reserves 20 lines for the CRTFORM and four lines for the TYPE area. If you specify a TYPE area of 2, the CRTFORM area increases to 22 lines.

If one procedure varies the size of the TYPE area from a larger to a smaller number, CRTFORM will clear the necessary TYPE statements in order to generate the next screen. If multiple CRTFORMs are written to the same screen, each CRTFORM should specify the same TYPE area size. For example:

```
CRTFORM LINE 1 TYPE 2
:
:
CRTFORM LINE 7 TYPE 2
```

Messages supplied with the HELPMESSAGE attribute in the Master File for fields on the MODIFY CRTFORM, are displayed in the TYPE area.

This type of message consists of one line of text which is displayed when:

- The value entered for a data source field is invalid according to the ACCEPT test for the field, or causes a format error.
- The user places the cursor in the data entry area for a particular field and presses a predefined PF key. If no message has been specified for that field, the following message will be displayed:

```
NO HELP AVAILABLE FOR THIS FIELD
```

Using FIDEL in Dialogue Manager

FIDEL works with all the standard Dialogue Manager facilities. However, the following differences apply when you use FIDEL with Dialogue Manager:

- You must allocate space for the variable field on the -CRTFORM, because variable fields in Dialogue Manager are not related to a Master File (see *Allocating Space on the Screen for Variable Fields* on page 10-78).
- There are two additional control statements: -CRTFORM BEGIN and -CRTFORM END. These give you control over when you begin and end the form (see *Starting and Ending CRTFORMS: BEGIN/END* on page 10-79). This control allows you to make use of other Dialogue Manager control statements as you are building your -CRTFORM.

Allocating Space on the Screen for Variable Fields

You must define the length of variable fields in -CRTFORMs. The length of Dialogue Manager variables can be defined in one of two ways:

- Directly on the -CRTFORM using the following syntax for allocating space.

```
<&variable/length
```

where:

```
length
```

Is a number representing the alphanumeric length of the variable.

- By using the -SET command to pre-declare the allocation of space using the syntax

```
-SET &variable = ' ' ;
```

where:

```
' '
```

Represents the alphanumeric length of the variable.

Note:

- If the variable format has been previously defined in the FOCEXEC procedure, the length defined directly on the -CRTFORM supersedes the previously defined format permanently.
- Variables used as label names (&:variable) cannot be automatically defined on the -CRTFORM. These variables must be defined with -SET statements.

Starting and Ending CRTFORMS: BEGIN/END

-CRTFORM BEGIN indicates that the form is being built. This Dialogue Manager control statement enables you to use other Dialogue Manager control statements between the screen lines without causing the CRTFORM to end. This is necessary when you are using indexed variables in a looping procedure.

-CRTFORM END terminates the form and causes the display of the assembled form.

Example Using Indexed Variables With -CRTFORM BEGIN and -CRTFORM END

The following is an example of the use of indexed variables in -CRTFORM. The variable &LINENUM is the indexed variable in the -CRTFORM. The index, &I, is set to increment by 1 each time a line is written. After the 10th line, the -CRTFORM ends. Note the use of the Dialogue Manager label, -BUILD and the -SET statement to control the loop within the form:

```

1.  -SET &I = 0;
2.  -CRTFORM BEGIN
    -"THE FOLLOWING FORM STORES 10 LINES OF TEXT"
    -" "
3.  -BUILD
4.  -SET &I = &I + 1;
5.  -SET &LINENUM.&I = 'LINE ' | &I;
6.  -"<D.&LINENUM.&I <&LINE.&I/60"
7.  -IF &I LT 10 GOTO BUILD;
8.  -CRTFORM END
    -*
    -TYPE LINE #2 CONTAINS THE FOLLOWING TEXT:
    -TYPE
9.  -TYPE &LINE2

```

This example processes as follows:

1. This -SET statement declares a counter, &I, and sets the counter to 0.
2. The -CRTFORM BEGIN statement begins the form.
3. This statement is a Dialogue Manager label, -BUILD. Because we are using the -CRTFORM BEGIN statement, this label does not end the CRTFORM.
4. This -SET statement sets the counter &I to increment by 1 each time a line is written. This controls the loop within the form.
5. This -SET statement indexes the variable &LINENUM with the counter &I. Thus, each time it is encountered in the -CRTFORM it will increment +1.

6. The -CRTFORM will appear as follows:

```
THE FOLLOWING FORM STORES 10 LINES OF TEXT  
  
LINE 1  
LINE 2  
LINE 3  
LINE 4  
LINE 5  
LINE 6  
LINE 7  
LINE 8  
LINE 9  
LINE 10
```

Type any text you wish onto the lines.

7. The -IF test allows the loop to process until there are 10 lines in the -CRTFORM. At that point control transfers to the -CRTFORM END statement.
8. -CRTFORM END ends the -CRTFORM and causes it to be displayed.
9. The last TYPE statement shows the contents of line 2.

Clearing the Screen in Dialogue Manager

The statement -CRTFORM both initiates the screen form and automatically clears the screen. The screen form begins at the top of the screen.

After the operator enters values for the variables and presses Enter, the variables are supplied with the values and the screen is cleared.

Changing the Size of the Message Area: -CRTFORM TYPE

By default, FOCUS reserves the last four lines of the Dialogue Manager terminal screen for TYPE messages. You can change this by using the keyword TYPE to determine the number of lines each CRTFORM reserves for messages. This feature allows you to increase the number of lines on the screen for CRTFORM display and reduce the number of lines reserved for messages at the bottom of the screen. The syntax is

```
-CRTFORM TYPE {n|4}
```

where:

n

Is a number from 1 to 4 indicating the number of message lines desired. The TYPE value setting remains in effect for all subsequent CRTFORMs in the same procedure until overridden by a new value. The default is 4.

You can expand the CRTFORM screen size by specifying a number less than 4. For example, a terminal with a height of 24 lines reserves 20 lines for the CRTFORM and four lines for the TYPE area. If you specify a TYPE area of 2, the CRTFORM area increases to 22 lines.

Annotated Example: -CRTFORM

The following FOCEXEC is an example of a TABLE request incorporating the use of -CRTFORM.

```

    -* Component Of Retail Sales Reporting Module
1.  SET &LIST = 'STAMFORD,UNIONDALE,NEWARK';
2.  PROMPT &CITY.(&LIST).ENTER CITY.:
    -*
3.  -CRTFORM
    -"Monthly Sales Report For <D.&CITY"
    -"Date: <D.&DATE      Time: <D.&TOD"
    -" "
    -"Beginning Product Code is:  <&BEGCODE/3"
    -"Ending Product Code is:     <&ENDCODE/3"
    -"Regional Supervisor is:     <&REGIONMGR/15"
    -"Title For UNIT_SOLD is:     <&UNIT_HEAD/10"

```

4. TABLE FILE SALES
HEADING CENTER
MONTHLY REPORT FOR &CITY"
"PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
SUM UNIT_SOLD AS &UNIT_HEAD
AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
IF CITY EQ &CITY
FOOTING CENTER
"REGION MANAGER: ®IONMGR"
"CALCULATED AS OF &DATE"
5. END

The following is a sample of the dialogue between the screen and the operator. Operator entries are in lowercase.

1. The -SET statement sets a value for the variable &LIST. The value is actually a list of the names of three cities. They are enclosed in single quotation marks because of the embedded commas.
2. The -PROMPT statement prompts the operator at the terminal for a value for &CITY. Assume the operator types a city that is not on the list:

```
ENTER CITY:  
boston  
PLEASE CHOOSE ONE OF THE FOLLOWING:  
STAMFORD, UNIONDALE, NEWARK  
ENTER CITY:  
stamford
```

3. The statement -CRTFORM initiates a screen form on which you type data:

```
Monthly Sales Report for STAMFORD  
Date: 01/08/2003           Time: 13.12.41  
  
Beginning Product Code is:   b10  
Ending Product Code is:     b20  
Regional Supervisor is:     smith  
Title For UNIT_SOLD is:     sales
```


4. The following are the stacked FOCUS commands as they appear on the FOCSTACK after the values have been entered from the -CRTFORM:

```
TABLE FILE SALES
HEADING CENTER
"MONTHLY REPORT FOR STAMFORD"
"PRODUCT CODES FROM B10 TO B20"
" "
SUM UNIT_SOLD AS SALES AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM B10 TO B20
IF CITY EQ STAMFORD
FOOTING CENTER
"REGION MANAGER: SMITH"
"CALCULATED AS OF 01/08/2003"
END
```

5. The report is as follows:

PAGE 1

MONTHLY REPORT FOR STAMFORD
PRODUCT CODES FROM B10 TO B20

PROD_CODE	SALES	RETURNS	RATIO
B10	60	10	16.67
B12	40	3	7.50
B17	29	2	6.90

REGION MANAGER: SMITH
CALCULATED AS OF 11/04/03

Using the FOCUS Screen Painter

The FOCUS Screen Painter allows you to design a FIDEL full-screen layout by placing literal text and areas for fields on the screen in any position that you desire. You then assign these field areas of the screen to a data source or computed fields, and FOCUS automatically codes the CRTFORM. You can also color, highlight, and/or assign screen attributes to sections of the screen (text, fields, background or any combination).

The FOCUS Screen Painter also allows you to generate CRTFORMs automatically without specifying field names (see *Generating Automatic CRTFORMs* on page 10-47).

The Screen Painter operates within TED, the FOCUS editor (see the *Overview and Operating Environments* manual for more details on TED), and can be used to create both MODIFY CRTFORMs and Dialogue Manager -CRTFORMs. It is easy to use and makes the creating of forms simple and visual.

Entering Screen Painter

To create a CRTFORM using the Screen Painter, you first enter the PAINT command from within TED. You can set up the PAINT screen as follows:

1. Enter TED by typing TED followed by the name of the file:

In CMS:	TED CRTEMP FOCEXEC
In MVS:	TED FOCEXEC (CRTEMP)

Either command will bring you into the FOCEXEC called CRTEMP. The FOCEXEC may or may not already exist.

2. Place a CRTFORM or -CRTFORM statement in the FOCEXEC if it is not already there. For example:

```
MODIFY FILE EMPLOYEE  
CRTFORM
```

3. When a FOCEXEC is on the screen, enter the PAINT command in the command area or press PF4. TED searches from the current line down the file until it finds a CRTFORM statement and makes the following line the current line. (If you use more than one CRTFORM in the FOCEXEC and you want to create the second CRTFORM, enter the command PAINT 2.)

Note: A Master File must be active for the Screen Painter to set the default field lengths for data source fields.

The following PAINT screen is displayed on your terminal:

```

.....1.....2.....3.....4.....5.....6.....7.....
.....1.....2.....3.....4.....5.....6.....7.....
COMMAND: _
01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
    
```

Between the two scale lines are 20 blank lines in which to enter the screen layout. The cursor is positioned in the command zone in the lower left portion of the screen. The codes at the bottom of the screen identify some of the PF keys that you can use.

These perform the following functions:

PF Key	Function
01=HELP	Lists all the PF key functions.
03=END	Transfers you from the PAINT screen back into TED, within your file.
07=BACKWARD	Scrolls back to the previous screen of the CRTFORM. When used with ASSIGN, moves the cursor back to the first field.
08=FORWARD	Scrolls forward to the next screen of the CRTFORM. When used with ASSIGN, moves the cursor to the next field.
09=ASGN-FLD	Use on the ASSIGN screen. Transfers you to the particular field that the cursor is placed on. You can then immediately assign or change attributes for that field.
10=ASSIGN	Transfers you from the PAINT screen to the ASSIGN screen (see <i>Identifying Fields: ASSIGN</i> on page 10-93).
11=FIDEL	Shows you the CRTFORM as it will appear on the screen.
17=BOX	Enables you to define a box of text. Move the cursor to the upper-left corner and press PF17. Select features from the box menu and then move the cursor to the bottom-right corner and press PF17.

Note: With the exception of FORWARD, BACKWARD and ASGN-FLD, you can also accomplish these functions by typing the command name in the command zone.

4. If the CRTFORM already includes fields, and one or more fields are not declared in the Master File, you may see this message:

(FOC532) LENGTHS OF FIELDS IN THIS CRTFORM CANNOT BE DETERMINED

To continue type IGNore and provide the lengths explicitly, or type ?F filename to activate the appropriate master. After you follow the message instructions, the PAINT screen appears.

PF Keys in PAINT

You can alter the values of PF keys in PAINT with the command

```
SET PFnn word
```

where:

nn

Is a number from 1 to 24 specifying the PF key to be set.

word

Is the new value for the key.

The initial PF key settings in PAINT are:

PF Key	Setting
PF1, PF13	: HELP
PF2, PF14	: INSERT
PF3, PF15	: END
PF4	: PAINT
PF5	: TOP
PF6	: BOTTOM
PF7, PF19	: BACKWARD PAGE
PF8, PF20	: FORWARD PAGE
PF9	: ASSIGN FIELD
PF10	: ASSIGN
PF11	: FIDEL
PF12	: DUPLICATE
PF16	: QUIT
PF17	: BOX
PF18	: (currently not used)

PF Key	Setting
PF21	: CRTFORM
PF22	: SET OUTPUT FIDEL
PF23	: SET OUTPUT DIALOGUE
PF24	: (currently not used)

Entering Data Onto the Screen

In PAINT, you may enter text, and specify field dimensions. Always use the arrow keys to designate text and field areas on this screen. Generally, text is entered by positioning the cursor and typing, but fields require type and width specifications.

To create a field, type

```
<xx . . . x
```

where the total number of x's equals the width of the field desired. If you do not specify a width, or if the command you entered is not syntactically correct, or active, PAINT will automatically default to a width defined in the Master File.

Fields are conditional by default. To specify non-conditional fields, enter

```
<xx . . . x>
```

where the total number of x's equals the width of the field.

You may enter text descriptions of each field, but do not type the field name after the left or right caret. Later you will learn how to assign each field a field name. You may designate the field as Entry, Turnaround or Display with the ASSIGN command (see *Identifying Fields: ASSIGN* on page 10-93). By default, the fields are conditional. To specify non-conditional, type a right caret (>) after the x's that indicate the field. We recommend that turnaround fields be non-conditional. (See *Conditional and Non-Conditional Fields* on page 10-42 for information on conditional and non-conditional fields.)

Editing Functions

When you are designing your screen, you have editing functions available to you. To use them, you must enter the command name on the COMMAND line on your PAINT screen or use the appropriate PF key:

- **Inserting Lines:** INSERT, PF2, PF14. You can insert lines by moving the cursor to any character on a line. Press PF2 or PF14 and the new line will be inserted immediately following the line where the cursor is positioned. If you want to insert more than one line, type the command (do not press Enter)

`I [INSERT] n`

where *n* is the number of new lines to be inserted. Next, move the cursor to the line where you want the lines inserted. Press Enter and *n* lines will be inserted beneath the line where the cursor is currently positioned.

If the insert causes the screen to exceed 20 lines, the message

`1, 40`

will be displayed, indicating that the display starts at line 1 out of a total of 40.

- **Deleting Lines:** DELETE. You can similarly delete lines by typing:

`D [DELETE] n`

on the command line, where *n* is the number of lines you want deleted. Next, move the cursor to the first line you want deleted and press Enter.

- **Duplicating Lines:** DUPLICATE, PF12. You can duplicate lines by placing the cursor on the line that you want to duplicate. Press PF12. If you want to duplicate more than one line, type the command

`DUPLICATE n`

where *n* is the number of copies you want; position the cursor on the line you want to duplicate and press Enter.

If the line that you are copying contains subscripted fields (for example, "SALES (1)"), the subscripts will be incremented by one automatically (see *Specifying Groups of Fields* on page 10-59). If you want an increment other than 1, enter the command

`DUPLICATE n m`

where *m* is the increment number.

Sample PAINT Screen

In the following example, assume that the following FOCEXEC exists:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID #: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CRTFORM
```

To use the Screen Painter to create the second CRTFORM, specify PAINT 2 at the TED command line (2 indicates second CRTFORM). Then type the following text and fields on the PAINT screen to create the CRTFORM that will be displayed if there is a match on EMP_ID.

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
                                     EMPLOYEE UPDATE
EMPLOYEE ID #: <XXXXXXXXX          LAST NAME: <XXXXXXXXXXXXXXXXXX
DEPARTMENT: <XXXXXXXXXX>          CURRENT SALARY: <XXXXXXXXX
BANK: <XXXXXXXXXXXXXXXXXXXXXXXXX

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:_
01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
```


When you finish entering text and indicating areas for fields (the number of X's corresponds to the field length), press Enter. The following screen results:

```

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
                                     EMPLOYEE UPDATE
EMPLOYEE ID #: <1111111111          LAST NAME: <22222222222222
DEPARTMENT: <1111111111>          CURRENT SALARY: <22222222
BANK: <11111111111111111111111111

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND: _
01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX

```

Note that the X's are replaced with numbers indicating the relative position of each field on a line. On the second line, EMPLOYEE ID is number 1 and LAST NAME is number 2.

Note: Labels created in Screen Painter cannot exceed 12 characters.

Identifying Fields: ASSIGN

Until now, you have simply laid out text that describes the fields, designated a display length (X's) within the left caret (<), and possibly indicated non-conditional (>) fields. Now you can assign field names and attributes for the fields. Enter the command ASSIGN in the command zone or press PF10. Your ASSIGN screen displays the following:

```

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: *****          LAST NAME: EEEEEEEEEEEEEEE
DEPARTMENT: EEEEEEEEEEE        CURRENT SALARY: EEEEEEEEE
BANK: EEEEEEEEEEEEEEEEEEE

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
Field:                          Entry/Turn Disp (E,T,D):   Col (W,B,R,P,G,A,Y):
Field Length: 9 (D12.2M) High/Nodis/Inv (H,N,I):         Label:

```

The first field following the descriptive text EMPLOYEE ID #: is highlighted and replaced by asterisks. All other fields are displayed in low intensity with E's denoting the length of the fields. The cursor is positioned in the status entry area at the bottom of the screen next to FIELD.

Now you can enter and assign field names and attributes for the field appearing in asterisks. Fill in the appropriate values in the status entry area at the bottom of the screen. To move from one status area to the next, press TAB. You may leave a blank where you do not want to use a particular attribute.

FIELD:

Enter the field name for the first field. In this case, enter EMP_ID, which is the name of the field in the Master File.

ENTRY/TURN/DISP (E,T,D):

You may designate the field as Entry, Turnaround, or Display by specifying E, T, or D, respectively. The default is Entry. (See *Data Entry, Display and Turnaround Fields* on page 10-14 for more information on Entry, Turnaround, and Display fields.) You specify whether a field is conditional or non-conditional when you enter the field on the PAINT screen (see *Entering Data Onto the Screen* on page 10-88).

COL (W,B,R,P,G,A,Y) :

You may designate the field with a color by entering one of the color abbreviations in the COL area. You may choose W, white; B, blue; R, red; P, pink; G, green; A, aqua; Y, yellow. If you do not wish to assign a color, leave this area blank.

FIELD LENGTH: 9 (A9) :

In MODIFY, if a Master File is active while you are assigning attributes, the LENGTH status will contain two values: the first value is the number of X's from the PAINT screen, which is the display value; the value in parentheses is the format value from the Master File. The display value must be equal to or less than the format value.

If you want to change the display value on the screen, put a new number in the FIELD LENGTH area or return to PAINT (PF3) and enter the correct number of characters following the <.

HIGH/NODISP/INV (H,N,I) :

You can choose highlight, nodisplay or inverse video as an attribute for the field by filling in the appropriate abbreviation.

LABEL :

If you want to enter a label, simply enter its name. The colon and period are automatically provided on the screen.

In the following example, the current field is LAST_NAME. It is designated a display field. The remaining attributes are left blank. After you press Enter and move to the next field, the asterisks turn to D's (display) as did the EMP_ID field.

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
                                     EMPLOYEE UPDATE
EMPLOYEE ID #: DDDDDDDDD                LAST NAME: *****
DEPARTMENT: EEEEEEEEEEE                CURRENT SALARY: EEEEEEEEEEEEEEE
BANK: EEEEEEEEEEEEEEEEEEEEEEE

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
FIELD: last_name          ENTRY/TURN/DISP (E,T,D): d    COL (B,R,P,G,A,Y) :
FIELD LENGTH: 15 (A,15) HIGH/NODISP/INV (H,N,I):      LABEL:
```

To move to the next field, press PF8. You may assign a field name, prefix, color, attribute or label to the remaining fields on the screen. If you need to move to a previous field to change something, press PF7. This will return you to the first field. From there you can use the TAB key to move to the field that you need.

To move to a specific field directly from PAINT or from within ASSIGN, place the cursor on that field and press PF9, ASGN-FLD.

Viewing the Screen: FIDEL

From the PAINT or ASSIGN screen, you can view the exact FIDEL screen that you have created. Press PF11 or type FIDEL in the command zone. As the following screen shows, all entry fields are blank and ready to receive data; all turnaround fields contain T's and may be typed over; all display fields contain D's and are protected:

```

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
                                     EMPLOYEE UPDATE
EMPLOYEE ID #: DDDDDDDDD          LAST NAME: DDDDDDDDDDDDDDD
DEPARTMENT: TTTTTTTTTT          CURRENT SALARY:
FIDEL: Press PF3 or PF15 to return to the PAINT screen.
    
```

As indicated on the FIDEL screen, to return to the PAINT screen press PF3 or PF15.

Generating CRTFORMs Automatically

To generate CRTFORMs automatically (that is, without specifying individual fields) from the FOCUS Screen Painter, use the asterisk (*) with CRTFORM in the PAINT screen command zone. (See *Generating Automatic CRTFORMs* on page 10-47 for information on CRTFORM * variations and syntax.)

The text description identifying field is the field name from the Master File. Key fields automatically become entry fields, and all other fields become turnaround fields. With multi-segment data sources, the CRTFORM * command ignores all segments following the first cross-reference (segment type KU or KM) described in the Master File.

For example, to generate a CRTFORM containing all fields in the EMPLOYEE Master File, do the following:

1. Type a MODIFY and a CRTFORM statement in a FOCEXEC.
2. Enter PAINT on the TED command line to invoke the Screen Painter.
3. Type CRTFORM * in the Screen Painter command zone.

The following PAINT screen results:

```

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

EMP_ID           :<11111111>      :
LAST_NAME        :<1111111111111111  :      FIRST NAME  :<2222222222:
HIRE_DATE        :<111111      :      DEPARTMENT :<2222222222:
CURR_SAL         :<111111111111  :      CUR_JOBCODE :<222      :
ED_HRS          :<111111      :
BANK_NAME        :<11111111111111111111  :
BANK_CODE        :<111111      :      BANK_ACCT   :<2222222222:
EFFECT_DATE      :<111111      :
DAT_INC          :<111111>      :
PCT_INC          :<111111      :      SALARY      :<222222222222:
JOBCODE          :<111      :
TYPE             :<1111>      :
ADDRESS_LN1      :<11111111111111111111  :
ADDRESS_LN2      :<11111111111111111111  :
ADDRESS_LN3      :<11111111111111111111  :
ACCTNUMBER       :<1111111111      :
PAY_DATE         :<111111>      :
GROSS            :<111111111111  :
DED_CODE         :<1111>      :
PF8=NEXT SCREEN PF7=PREVIOUS SCREEN PF1=OUT

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:                                               1, 40
01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX

```

CRTFORM * creates labels (that is, text describing each field) on the CRTFORM of up to 12 characters. If the field name is shorter than 12 characters, the label is the field name. If the field name exceeds 12 characters, a caret (^) in the 12th position indicates a longer field name.

Terminating Screen Painter

To return to TED from the PAINT screen, enter the command END in the command zone or press PF3 until the prompt for TED appears. TED displays the lines as they have been generated, beginning at the current line, which is ON MATCH CRTFORM:

```

" <.C.                EMPLOYEE UPDATE                <0X
" <.C.                <0X
      <.C. "
" <.C. EMPLOYEE ID #: <D.EMP_ID/09                LAST NAME:    <0X
<LAST_NAME/15      <.C. "
" <.C.                <0X
      <.C. "
" <.C. DEPARTMENT: <T.DEPARTMENT/10>                CURRENT SALARY: <0X
<T.CURR_SAL/08      <.C. "
" <.C.                <0X
      <.C. "
" <.C. BANK <T.BANK_NAME/20                <.C. "
" <.C.                <0X
DATA
END

```

The generated code for the CRTFORM is in the file. Notice that each field is named and has its length appended to it. Any attributes or labels requested during the ASSIGN process are also present. If you want to change the layout, you can use the TED editor or you can return to the PAINT and/or ASSIGN screen to make the changes.

You can add further MATCH logic to the FOCEXEC by using TED. For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID #: <EMP_ID"
  MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH CRTFORM
"      EMPLOYEE UPDATE"
" "
" EMPLOYEE ID #: <D.EMP_ID/09                LAST NAME: <D.LAST_NAME/15"
" "
" DEPARTMENT: <:FIRST.H.T.DEPARTMENT/10> CURRENT SALARY: <0X
<.C.CURR_SAL/08"
" "
" BANK : <BANK_NAME/20"
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON MATCH CONTINUE TO BANK_NAME
  ON NOMATCH INCLUDE
  ON MATCH REJECT
DATA
END
```

If you want to add another CRTFORM screen at this point, make sure you are on the current line, type the CRTFORM or -CRTFORM statement, and reenter PAINT to design the next screen. Finally, you can exit the PAINT screen, return to TED, and add or change further logic.

Alternatively, all of the logic of the request could have been entered first and then the Screen Painter used to create all the FIDEL screens. To create the first screen, enter the command PAINT or PAINT 1; to create the second screen, enter the command PAINT 2. PAINT 2 locates the second CRTFORM statement starting from the current line. You can continue with PAINT 3, and so on, for all subsequent CRTFORM statements in the procedure.

CHAPTER 11

Creating and Rebuilding Databases

Topics:

- Creating New Databases: The CREATE Command
- Rebuilding Databases: The REBUILD Command
- Optimizing File Size: The REBUILD Subcommand
- Changing Database Structure: The REORG Subcommand
- Indexing Fields: The INDEX Subcommand
- Creating an External Index: The EXTERNAL INDEX Subcommand
- Checking Database Integrity: The CHECK Subcommand
- Changing the Database Creation Date and Time: The TIMESTAMP Subcommand
- Converting Legacy Dates: The DATE NEW Subcommand
- Migrating to a Fusion Database: The MIGRATE Subcommand
- Creating a Multi-Dimensional Index: The MDINDEX Subcommand

You can create a new database, or reinitialize an existing database, using the CREATE command.

Once a database exists, you may find it necessary to reorganize it in order to use disk space more effectively; to change the contents, index, or structure of the database; to change legacy date fields to smart date fields; or to convert a FOCUS database to a Fusion database. You can do all of this and more using the REBUILD command.

You can use the CREATE and REBUILD commands with FOCUS databases. You can also use CREATE to create relational tables for which you have the appropriate Interface.

Creating New Databases: The CREATE Command

You can create a new, empty FOCUS database for a Master File using the CREATE command. You can also use CREATE to erase the data in an existing FOCUS database.

CREATE also works for Fusion databases and, with the appropriate Interface installed, a relational table (such as a DB2 or Teradata table). For information, see the documentation for the relevant FOCUS Interface.

Note that on CMS, CREATE is not the only way to create a new FOCUS or Fusion database: you can also do so by running a MODIFY request that specifies the Master File of the database you wish to create.

Syntax **How to Use the CREATE Command**

The syntax of the CREATE command is

```
CREATE FILE mastername
```

where:

```
mastername
```

Is the name of the Master File that describes the database.

After you enter the CREATE command, FOCUS displays

```
NEW FILE fileid ON date AT time
```

where:

```
fileid
```

Is the complete file ID of the new database. For CMS, it is *filename filetype filemode*. For MVS, it is the ddname to which the source is allocated.

```
ON date AT time
```

Is the date and time at which the database was created or recreated.

When you issue the CREATE command, if the database already exists, FOCUS displays this message:

```
(FOC441) WARNING. THE FILE EXISTS ALREADY. CREATE WILL WRITE OVER IT  
REPLY :
```

To erase the database and create a new, empty database, enter Y. To cancel the command and leave the database intact, enter END, QUIT, or N.

If you wish to give the database absolute File Integrity protection (discussed in the *Developing Applications* manual), issue the following command prior to the CREATE command:

```
SET SHADOW=ON
```

Note: IBM no longer guarantees that CMS file mode A6 ensures absolute file integrity.

Note the following when issuing CREATE on MVS:

- If you do not allocate the database prior to issuing the CREATE command, FOCUS creates the database as temporary data set. To retain the database, copy it to a permanent data set with the TSO COPY or DYNAM COPY command.
- The CREATE command pre-formats the primary space allocation and initializes the database entry in the File Directory Table. A Master File must exist for the database in a PDS allocated to ddname MASTER.
- Issuing MODIFY or Maintain commands against uninitialized databases (those for which no CREATE was issued) results in a read error.

Example Recreating a FOCUS Database in CMS

To recreate the CAR database, issue the following command:

```
CREATE FILE CAR
```

FOCUS responds:

```
(FOC441) WARNING. THE FILE EXISTS ALREADY. CREATE WILL WRITE OVER IT
REPLY :
```

You would reply:

```
YES
```

FOCUS responds:

```
NEW FILE CAR      FOCUS   A1 ON 03/02/1999 AT 15.48.57
```

The CAR database still exists on disk, but it contains no records.

Example Creating a FOCUS Database in MVS

To create the ADDRESS database, allocate the database and then issue the CREATE command:

```
DYNAM ALLOC F(ADDRESS) DA(ADDRESS.FOCUS) NEW SPACE(5,5) CYL
CREATE FILE ADDRESS
```

FOCUS responds:

```
NEW FILE ADDRESS          ON 03/02/1999 AT 15.16.59
```

This creates the new FOCUS database ADDRESS.FOCUS allocated to ddname ADDRESS.

Rebuilding Databases: The REBUILD Command

You can make a structural change to a FOCUS or Fusion database after it has been created using the REBUILD command. Using REBUILD and one of its nine subcommands REBUILD, REORG, INDEX, EXTERNAL INDEX, CHECK, TIMESTAMP, DATE NEW, MDINDEX, and MIGRATE you can:

- Rebuild a disorganized database (REBUILD).
- Delete instances according to a set of screening conditions (REBUILD, or REORG).
- Redesign an existing database. This includes adding and removing segments, adding and removing data fields, indexing different fields, changing the size of alphanumeric data fields and more (REORG).
- Index up to seven new fields before rebuilding the database (INDEX).
- Create an external index database that facilitates indexed retrieval when joining or locating records (EXTERNAL INDEX).
- Check the structural integrity of the database (CHECK). Determine when the FOCUS database was last changed (TIMESTAMP).
- Convert legacy date formats to smart date formats (DATE NEW).
- Build or modify a multi-dimensional index for Fusion databases (MDINDEX).
- Convert FOCUS Master Files and databases to Fusion Master Files and databases (MIGRATE).

Note: Attempting to convert an alpha field to AnV, or vice versa, yields the following error:

```
(FOC 928) INVALID CHANGE IN FIELD FORMAT: In REBUILD you are not
allowed to change the field format; you are only allowed to change the
format width (e.g., A10 to A16).
```

You can use the REBUILD facility:

- **Interactively at the screen**, by issuing the REBUILD command at the FOCUS command prompt.
- **As a batch procedure**, by entering the REBUILD command, the desired subcommand, and any responses to subcommand prompts on separate lines of a procedure.

Before using the REBUILD facility, you should be aware of several required and recommended prerequisites regarding file allocation, security authorization, and backup.

Reference **Before You Use REBUILD: Prerequisites**

Before you use the REBUILD facility, there are several prerequisites that you need to consider:

- **Partitioning.** You can only REBUILD one partition of a partitioned FOCUS data source at one time. You must explicitly allocate the partition you want to REBUILD. Alternatively, you can create separate Master Files for each partition. For information about partitioned FOCUS data sources, see the *Describing Data* manual.
- **Size.** To REBUILD a FOCUS data source that is larger than one gigabyte on MVS, you must explicitly allocate ddname REBUILD to a temporary file with enough space to contain the data; on VM you must have enough TEMP space available. It is strongly recommended that you REBUILD/REORG to a new file, in sections, to avoid the need to allocate large amounts of space to REBUILD. In the DUMP phase, use selection criteria to dump a section of the database. In the LOAD phase, make sure to *add* each new section after the first. To add to a database in MVS, you must issue the LOAD command with the following syntax:

```
LOAD NOCREATE
```

- **Allocation.** Usually, you do not have to allocate work space prior to using a REBUILD command; FOCUS issues its own commands for this purpose. (To see if this is true for your operating system, see the *Overview and Operating Environments* manual.) However, adequate work space, such as temporary attached disk storage, must be available. As a rule of thumb, have space 10 to 20% larger than the size of the existing file available for the REBUILD and REORG options.

FOCUS always assigns the file name REBUILD to the work space. In the DUMP phase of the REORG command, the allocation statement is displayed in case you want to run the LOAD phase at a different time.

Also, if you are using the REBUILD facility interactively, you must allocate SYSPRINT to the terminal in order to view the menu. For more information on using SYSPRINT, see *Overview and Operating Environments*.

- **Security authorization.** If the database you are rebuilding is protected by a database administrator, you must be authorized for read and write access in order to perform any REBUILD activity. For more information on database security, see the *Describing Data* manual.
- **Backup.** Although it is not a requirement, we recommend that you create a backup copy of the original Master File and database before using any of the REBUILD subcommands.

Procedure How to Use the REBUILD Facility Interactively

Before you can use the REBUILD facility interactively, you must allocate SYSPRINT to the terminal. For more information on using SYSPRINT, see *Overview and Operating Environments*.

To invoke REBUILD interactively, issue

```
REBUILD
```

at the FOCUS command prompt. A menu of subcommands appears. Choose an option by entering either the subcommand or its corresponding number. If you enter a number, REBUILD displays the corresponding subcommand name. If you select the wrong subcommand you can enter QUIT to exit.

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. Timestamp | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smart date formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

The subsequent prompts depend on the subcommand you selected. Generally, you will only need to give the name of the database and possibly one or two other items of information.

Controlling the Frequency of REBUILD Messages

When REBUILD processes a database, it displays status messages periodically (for example, REFERENCE...AT SEGMENT 1000) to inform you of the progress of the rebuild. The default display interval is every 1000 segment instances processed during REBUILD's database retrieval and load phases. The number of messages displayed is determined by the number of segment instances in the FOCUS database being rebuilt, divided by the display interval.

Under CMS, the number of messages displayed for larger FOCUS databases can be problematic because of limited CMS spool space. You can control the number of messages displayed by setting the display interval using the SET REBUILDMSG command.

Syntax **How to Control the Frequency of REBUILD Messages**

REBUILD displays a message (REFERENCE...AT SEGMENT *segnum*) at periodic intervals to inform you of its progress as it processes a database. You can control the frequency with which REBUILD displays this message by issuing the command

```
SET REBUILDMSG = {n|1000}
```

where:

n

Is any integer from 1,000 to 99,999,999.

A setting of less than 1000:

- Generates a warning message that describes the valid values (0 or greater than 999).
- Keeps the current setting. The current setting will either be the default of 1000, or the last valid integer greater than 999 that REBUILDMSG was set to. A setting of 0 disables the message.

Example **Controlling the Display of REBUILD Messages**

The following example shows a REBUILD CHECK function where REBUILDMSG has been set to 4000, and the database contains 19,753 records.

Enter option

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smart date formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

CHECK

ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)

...

STARTING..

```
REFERENCE..AT SEGMENT    4000
REFERENCE..AT SEGMENT    8000
REFERENCE..AT SEGMENT   12000
REFERENCE..AT SEGMENT   16000
NUMBER OF SEGMENTS RETRIEVED= 19753
CHECK COMPLETED...
```

Optimizing File Size: The REBUILD Subcommand

You use the REBUILD subcommand for one of two reasons. Primarily, you use it to improve data access time and storage efficiency. After many deletions, the physical structure of your data does not match the logical structure. REBUILD REBUILD dumps data into a temporary work space and then reloads it, putting instances back in their proper logical order. A second use of REBUILD REBUILD is to delete segment instances according to a set of screening conditions.

Normally, you use the REBUILD subcommand as a way of maintaining a clean database. To determine whether you need to rebuild your database, enter the ? FILE command (described in *Confirming Structural Integrity Using ? FILE and TABLEF* on page 11-36):

```
? FILE filename
```

If your database is disorganized, the following message appears:

```
FILE APPEARS TO NEED THE -REBUILD-UTILITY  
REORG PERCENT IS A MEASURE OF FILE DISORGANIZATION  
0 PCT IS PERFECT -- 100 PCT IS BAD  
REORG PERCENT x%
```

This message appears whenever the REORG PERCENT measure is more than 30%. The REORG PERCENT measure indicates the degree to which the physical placement of data in the database differs from its logical, or apparent, placement.

The &FOCDISORG variable can be used immediately after the ? FILE command and also shows the percentage of disorganization in a database. &FOCDISORG will show a database's percentage of disorganization even if it is below 30% (see the *Developing Applications* manual).

Procedure How to Use the REBUILD Subcommand

The following steps describe how to use REBUILD's REBUILD subcommand:

1. At the FOCUS command prompt enter:

```
REBUILD
```

FOCUS displays the prompt:

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

2. Select the REBUILD option by entering:

```
REBUILD or 1
```

FOCUS prompts you to:

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

3. Enter the file ID of the database to be rebuilt:

```
fileid
```

FOCUS prompts you:

```
ANY RECORD SELECTION TESTS? (YES/NO)
```

4. If you are simply rebuilding the database, enter:

NO

FOCUS will immediately begin the REBUILD procedure (go to step 5).

On the other hand, if you wish to place screening conditions on the REBUILD subcommand, enter:

YES

FOCUS replies:

```
ENTER SELECTION TESTS (END LAST LINE WITH , $ )
```

The syntax for the selection test is the same as for the LOCATE subcommand of the FOCUS FSCAN facility (see *How to Locate an Instance Based on Field Values: The LOCATE Command* in Chapter 13, *Directly Editing FOCUS Databases With FSCAN*). Test relations of EQ, NE, LE, GE, LT, GT, CO (contains), and OM (omits) are permitted. Tests are connected with the word AND, and lists of literals may be connected with the OR operator. A comma followed by a dollar sign (,\$) is required to terminate any test.

For example, you might enter the following:

```
A EQ A1 OR A2 AND B LT 100 AND  
C GT 400 AND D CO 'CUR', $
```

5. After the REBUILD procedure begins, you will see a screen similar to the following when the procedure is complete

```
STARTING..  
FILEDEF REBUILD DISK REBUILD FOCTEMP A4 (LRECL 00088 BLKSIZE 08804  
NUMBER OF SEGMENTS RETRIEVED = n  
NEW FILE fileid ON date AT time  
NUMBER OF SEGMENTS INPUT = m  
FILE HAS BEEN REBUILT
```

where:

n

Is the number of segments retrieved.

fileid

Is the file ID of the database being rebuilt.

date

Is the current date.

time

Is the current time.

m

Is the number of segments input into the rebuilt database.

Example Using the REBUILD Subcommand in CMS

```

rebuild
Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX             (Build/modify the database index)
 4. EXTERNAL INDEX   (Build/modify an external index database)
 5. CHECK             (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)

1
REBUILD
ENTER NAME OF DATABASE FILE (FN FT FM)
employee focus a
ANY RECORD SELECTION TESTS? (YES/NO)
no
STARTING..
FILEDEF REBUILD DISK      REBUILD FOCTEMP A4          (LRECL  00088  BLKSIZE 23940
RECFM  VB                )
NUMBER OF SEGMENTS RETRIEVED=      582

NEW FILE EMPLOYEEFOCUS  A1 ON 05/10/1999 AT 15.49.27
NUMBER OF SEGMENTS INPUT=      582
FILE HAS BEEN REBUILT

```

Example Using the REBUILD Subcommand in MVS

```
rebuild

Enter option
 1. REBUILD          (Optimize the database structure)
 2. REORG            (Alter the database structure)
 3. INDEX            (Build/modify the database index)
 4. EXTERNAL INDEX  (Build/modify an external index database)
 5. CHECK            (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
rebuild

ENTER NAME OF DATABASE FILE
> employee

ANY RECORD SELECTION TESTS? (YES/NO)
> no
STARTING..
  DCB USED WITH FILE REBUILD IS DCB=(RECFM=VB,LRECL=00088,BLKSIZE=23940)
  NUMBER OF SEGMENTS RETRIEVED=      576
  NEW FILE EMPLOYEE ON 05/14/1999 AT 09.31.26
  NUMBER OF SEGMENTS INPUT=      576
  FILE HAS BEEN REBUILT
```

Changing Database Structure: The REORG Subcommand

The REORG subcommand enables you to make a variety of changes to the FOCUS Master File *after* data has been entered in the FOCUS database. REBUILD REORG is a two-step procedure that first dumps FOCUS data into a temporary work space and then reloads it under a new Master File.

You can use REBUILD REORG to:

- Add new segments as descendants of existing segments.
- Remove segments.
- Add new data fields at the end of existing segments.

Note: The fields must be added after the key fields.

- Remove data fields.
- Change the order of non-key data fields within a segment; key fields may not be changed.
- Promote fields from unique segments to parent segments.
- Demote fields from parent segments to descendant unique segments.
- Index different fields.
- Increase or decrease the size of an alphanumeric data field.

REBUILD REORG will not enable you to:

- Change field format types (alphanumeric to numeric and vice versa, changing numeric format types).
- Change the value for SEGNAME attributes.
- Change the value for SEGTYPE attributes.
- Change field names that are indexed.

To accomplish these tasks you must use FIXFORM. See Chapter 9, *Modifying Data Sources With MODIFY*, for more information.

Procedure How to Use the REORG Subcommand

The following steps describe how to use REBUILD's REORG subcommand under CMS:

1. Before making any changes to the original Master File, make a copy of it with another name.
2. Go into TED, or another editor, and make the desired edits to the copy of the Master File.
3. At the FOCUS prompt, initiate REBUILD by entering:

```
REBUILD
```

FOCUS displays the prompt:

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

4. Select the REORG option by entering:

```
REORG or 2
```

FOCUS responds:

```
Enter option
```

- | | |
|---------|---------------------------------|
| 1. DUMP | (DUMP contents of the database) |
| 2. LOAD | (LOAD data into the database) |

If you want to mount a scratch tape for work space during the DUMP phase, you can type the name of the tape after the word REORG.

5. Initiate the dump phase of the procedure by entering:

```
DUMP or 1
```

FOCUS displays:

```
DUMP
```

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

6. You are prompted for the file ID of the FOCUS database you wish to dump from. Be sure to use the name of the original Master File for this phase. Enter:

```
fileid
```

7. You are asked if you want selection tests on your data. If you do, answer YES; FOCUS will dump only data that meets your specifications (See *How to Locate an Instance Based on Field Values: The LOCATE Command* in Chapter 13, *Directly Editing FOCUS Databases With FSCAN*, for selection test syntax). It is more likely, however, that you will want to dump the entire database. To do so, answer:

NO

FOCUS displays the following as it dumps

```
STARTING..
FILEDEF REBUILD DISK REBUILD FOCTEMP A4 (LRECL 00088 BLKSIZE 23940
NUMBER OF SEGMENTS RETRIEVED=n
```

where:

n

Is the number of segments that were successfully dumped.

You should now see the FOCUS prompt:

>

8. You are ready to begin the second phase of REBUILD REORG: LOAD. At the FOCUS prompt, enter once again:

REBUILD

You will again see the prompt:

```
Enter option
1. REBUILD          (Optimize the database structure)
2. REORG            (Alter the database structure)
3. INDEX            (Build/modify the database index)
4. EXTERNAL INDEX  (Build/modify an external index database)
5. CHECK            (Check the database structure)
6. TIMESTAMP        (Change the database timestamp)
7. DATE NEW         (Convert old date formats to smartdate formats)
8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
```

9. Enter:

REORG or 2

You will see:

```
Enter option
1. DUMP              (DUMP contents of the database)
2. LOAD              (LOAD data into the database)
```

10. Specify the LOAD phase by entering:

LOAD or 2

FOCUS responds:

```
LOAD  
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

11. FOCUS is prompting you to enter the file ID of the database you are loading from the temporary file, REBUILD (see step 7). In most cases, this will be the new database name

fileid

where:

fileid

Is the name of the new database.

FOCUS responds

```
STARTING..  
NEW FILE fileid ON date AT time  
NUMBER OF SEGMENTS INPUT=n
```

where:

fileid

Is the file ID of the database that has received the data.

date

Is the current date.

time

Is the current time.

n

Is the number of segments successfully loaded.

At this stage, you have loaded the specified data from the original Master File into a new database with the file ID you specified. It is important to remember that both the Master File and database for the original Master File remain on your disk. You have three choices. You may:

- rename the new Master File and database to prevent possible confusion.
- rename the new Master and database to the original name. As a result, any existing FOCEXECs referencing the original name will run against the new database.
- delete the original Master and database after you verify that the new Master and database are correct and complete.

In CMS, if you enter the name of a database that already exists, (the original Master File) FOCUS prompts you that you will be appending data to a preexisting database and asks if you wish to continue.

In MVS, FOCUS doesn't ask you if you want to append to an existing database, it just creates the data source. If you want to append, when you issue the LOAD command, say LOAD NOCREATE.

Enter N to terminate REBUILD execution (do not enter NO). Enter Y to add the records in the temporary REBUILD file to the original FOCUS database.

If duplicate field names occur in a Master File, REBUILD REORG is not supported.

In MVS, you must issue a CREATE for a new database being loaded.

Example Using the REORG Subcommand in CMS

First make a copy of the database:

```
cms copy employee focus a oldemp focus a
```

Now start the DUMP phase:

```
rebuild
```

```
Enter option
```

- 1. REBUILD (Optimize the database structure)
- 2. REORG (Alter the database structure)
- 3. INDEX (Build/modify the database index)
- 4. EXTERNAL INDEX (Build/modify an external index database)
- 5. CHECK (Check the database structure)
- 6. TIMESTAMP (Change the database timestamp)
- 7. DATE NEW (Convert old date formats to smartdate formats)
- 8. MDINDEX (Build/modify a multidimensional index. FUSION DBs only)
- 9. MIGRATE (Convert FOCUS masters/DBs to FUSION)

```
2
```

```
Enter option
```

- 1. DUMP (DUMP contents of the database)
- 2. LOAD (LOAD data into the database)

```
dump
```

```
DUMP
```

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

```
employee focus a
```

```
ANY RECORD SELECTION TESTS? (YES/NO)
```

```
no
```

```
STARTING..
```

```
FILEDEF REBUILD DISK REBUILD FOCTEMP A4 (LRECL 00088 BLKSIZE 23940  
RECFM VB )  
NUMBER OF SEGMENTS RETRIEVED= 576  
>
```

Now erase the employee database and start the LOAD phase:

```
cms erase employee focus a
```

```
rebuild
```

```
Enter option
```

- 1. REBUILD (Optimize the database structure)
- 2. REORG (Alter the database structure)
- 3. INDEX (Build/modify the database index)
- 4. EXTERNAL INDEX (Build/modify an external index database)
- 5. CHECK (Check the database structure)
- 6. TIMESTAMP (Change the database timestamp)
- 7. DATE NEW (Convert old date formats to smartdate formats)
- 8. MDINDEX (Build/modify a multidimensional index. FUSION DBs only)
- 9. MIGRATE (Convert FOCUS masters/DBs to FUSION)

```
reorg
```

```
Enter option
```

- 1. DUMP (DUMP contents of the database)
- 2. LOAD (LOAD data into the database)

```
load
```

```
LOAD
```

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

```
employee focus a
```

```
STARTING..
```

```
NEW FILE EMPLOYEEFOCUS A ON 05/10/1999 AT 16.34.17
```

```
NUMBER OF SEGMENTS INPUT= 576
```

```
>
```

Example Using the REORG Subcommand in MVS

First make a copy of the database:

```
dynam copy employee.focus oldemp.focus
```

Now start the DUMP phase:

```
rebuild
```

Enter option

- 1. REBUILD (Optimize the database structure)
- 2. REORG (Alter the database structure)
- 3. INDEX (Build/modify the database index)
- 4. EXTERNAL INDEX (Build/modify an external index database)
- 5. CHECK (Check the database structure)
- 6. TIMESTAMP (Change the database timestamp)
- 7. DATE NEW (Convert old date formats to smartdate formats)
- 8. MDINDEX (Build/modify a multidimensional index. FUSION DBs only)
- 9. MIGRATE (Convert FOCUS masters/DBs to FUSION)

```
reorg
```

Enter option

- 1. DUMP (DUMP contents of the database)
- 2. LOAD (LOAD data into the database)

```
dump
```

```
DUMP
```

```
ENTER NAME OF FOCUS/FUSION FILE
```

```
> employee
```

```
ANY RECORD SELECTION TESTS? (YES/NO)
```

```
> no
```

```
STARTING..
```

```
DCB USED WITH FILE REBUILD IS DCB=(RECFM=VB,LRECL=00088,BLKSIZE=23940)
```

```
NUMBER OF SEGMENTS RETRIEVED= 576
```

Now start the LOAD phase:

```
> > rebuild
Enter option
 1. REBUILD          (Optimize the database structure)
 2. REORG           (Alter the database structure)
 3. INDEX           (Build/modify the database index)
 4. EXTERNAL INDEX  (Build/modify an external index database)
 5. CHECK           (Check the database structure)
 6. TIMESTAMP       (Change the database timestamp)
 7. DATE NEW        (Convert old date formats to smartdate formats)
 8. MDINDEX         (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE         (Convert FOCUS masters/DBs to FUSION)
> reorg
Enter option
 1. DUMP            (DUMP contents of the database)
 2. LOAD            (LOAD data into the database)
LOAD
ENTER NAME OF FOCUS/FUSION FILE
> employee

STARTING..
  NEW FILE EMPLOYEE          ON 05/14/1999 AT 09.41.37
 NUMBER OF SEGMENTS INPUT=   576
> >
```

Indexing Fields: The INDEX Subcommand

To index a field after you have entered data into the database, use the INDEX subcommand. You can index up to seven fields in addition to those previously specified in the Master File or since the last REBUILD or CREATE command. The maximum number of indexes that a FOCUS database can support is documented in the *Describing Data* manual. The only requirement is that each field specified must be described with the FIELDTYPE=I (or INDEX=I) attribute in the Master File. If you add more than seven index fields, REBUILD INDEX yields the following diagnostic:

```
(FOC720) THE NUMBER OF INDEXES ADDED AFTER FILE CREATTION EXCEEDS 7
```

The INDEX option uses the operating system sort program. You must have disk space to which you can write. To calculate the amount of space needed, add 8 to the length of the index field in bytes and multiply the sum by twice the number of segment instances

```
(LENGTH + 8) * 2n
```

where:

n

Is the number of segment instances.

You may decide to wait until after loading data to add the FIELDTYPE=I attribute and index the field. This is because the separate processes of loading data and indexing can be faster than performing both processes at the same time when creating the database. This is especially true for large databases.

Sort libraries and work space must be available. The REBUILD allocates default sort work space in MVS, if you have not already done so. DDNAMEs must be allocated prior to issuing a REBUILD INDEX. SORTOUT and SORTLIB may also be needed.

In CMS, the following command identifies a specific sort product to FOCUS

```
SET SORTLIB = sortname
```

where:

sortname

Can be SYNCSORT, DFSORT, or VMSORT.

A CMS GLOBAL TXTLIB command must be issued prior to REBUILD INDEX to identify the location of the sort program. If the GLOBAL TXTLIB command and the SET SORTLIB commands are not issued, FOCUS will:

- Look for the sort program in SORTLIB TXTLIB.
- Automatically issue a GLOBAL TXTLIB SORTLIB command.

If SORTLIB TXTLIB is not available to FOCUS at the time of REBUILD INDEX, the following error occurs:

```
(FOC263) EXTERNAL FUNCTION OR LOAD MODULE NOT FOUND: SORT
```

Procedure How to Use the Index Subcommand

To use REORG INDEX, follow these steps:

1. Add the FIELDTYPE=I attribute to the field or fields you are indexing in the Master File.
2. Begin the REBUILD utility by entering at the FOCUS prompt:

```
REBUILD
```

FOCUS responds:

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

3. Invoke the INDEX option by entering:

```
INDEX or 3
```

FOCUS displays:

```
ENTER THE NAME OF THE MASTER
```

4. Enter the name of the Master File in which you will add the INDEX=I or FIELDTYPE=I attribute:

```
mastername
```

FOCUS then prompts you to:

```
ENTER NAME OF FIELD TO INDEX (OR * FOR ALL)=
```

5. If you are indexing only one field, enter the name of the field here. If you are indexing all the fields that have FIELDTYPE=I, enter:

*

FOCUS begins the procedure:

STARTING..

After completing the procedure, it displays the following

```
INDEX VALUES RETRIEVED=      n
SORT COMPLETE .. RET CODE    0
INDEX INITIALIZED FOR: fieldname
INDEX VALUES INCLUDED= n
```

where:

fieldname

Is the name of a field that was indexed.

n

Is the number of instances for that field.

Example Using the INDEX Subcommand in CMS

The following example illustrates REBUILD INDEX:

```
cms global txtlib vmslib
set sortlib = vmsort
rebuild
```

Enter option

1. REBUILD (Optimize the database structure)
2. REORG (Alter the database structure)
3. INDEX (Build/modify the database index)
4. EXTERNAL INDEX (Build/modify an external index database)
5. CHECK (Check the database structure)
6. TIMESTAMP (Change the database timestamp)
7. DATE NEW (Convert old date formats to smartdate formats)
8. MDINDEX (Build/modify a multidimensional index. FUSION DBs only)
9. MIGRATE (Convert FOCUS masters/DBs to FUSION)

index

ENTER THE NAME OF THE MASTER

employee

ENTER NAME OF FIELD TO INDEX (OR * FOR ALL)

employee

ENTER NAME OF FIELD TO INDEX (OR * FOR ALL)

emp_id


```

STARTING..
(FOC319) WARNING. THE FIELD IS INDEXED AFTER THE FILE WAS CREATED:
EMP_ID
INDEX VALUES RETRIEVED=      12
SORT COMPLETE .. RET CODE    0
INDEX INITIALIZED FOR: EMP_ID
INDEX VALUES INCLUDED=      12
>

```

Example Using the INDEX Subcommand in MVS

REBUILD INDEX uses an external sort. FOCUS searches for the system-installed sort product using its normal search path.

```

> > tso alloc f(sortout) sp(1 1) tracks
> > rebuild

Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX             (Build/modify the database index)
 4. EXTERNAL INDEX   (Build/modify an external index database)
 5. CHECK             (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs
only)
 9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
> 3
INDEX

ENTER THE NAME OF THE MASTER
> employee
ENTER NAME OF FIELD TO INDEX (OR * FOR ALL)
> emp_id
STARTING..
(FOC319) WARNING. THE FIELD IS INDEXED AFTER THE FILE WAS CREATED:
EMP_ID
INDEX VALUES RETRIEVED=      12
SORT COMPLETE .. RET CODE    0
INDEX INITIALIZED FOR: EMP_ID
INDEX VALUES INCLUDED=      12

```

Creating an External Index: The EXTERNAL INDEX Subcommand

Users with READ access to a local FOCUS database can create an index database that facilitates indexed retrieval when joining or locating records. An external index is a FOCUS database that contains index, field, and segment information for one or more specified FOCUS or Fusion databases. The external index is independent of its associated FOCUS or Fusion database. External indexes offer equivalent performance to permanent indexes for retrieval and analysis operations. External indexes enable indexing on concatenated FOCUS databases, indexing on real and defined fields, and indexing selected records from WHERE/IF tests. External indexes are created as temporary data sets unless pre allocated to a permanent data set. They are not updated as the indexed data changes.

You create an external index with the REBUILD command. Internally, REBUILD begins a process which reads the databases that make up the index, gathers the index information, and creates an index database containing all field, format, segment, and location information.

You provide information regarding:

- Whether you want to add new records from a concatenated database to the index database.
- The name of the external index database that you want to build.
- The name of the database from which the index information is obtained.
- The name of the field from which the index is to be created.
- Whether you want to position the index field within a particular segment.
- Any valid WHERE or IF record selection tests.

Sort libraries and work space must be available. The REBUILD allocates default sort work space in MVS, if you have not already done so. DDNAMEs SORTIN and SORTOUT must be allocated prior to issuing a REBUILD.

Procedure How to Use the EXTERNAL INDEX Subcommand

To create an external index from a concatenated database, follow these steps:

1. Assume that you have the following USE in effect:

```
USE CLEAR *
USE
EMPLOYEE
EMP2 AS EMPLOYEE
JOBFILE
EDUCFILE
END
```

Note that EMPLOYEE and EMP2 are concatenated and can be described by the EMPLOYEE Master File.

2. At the FOCUS prompt, initiate REBUILD by entering:

```
REBUILD
```

FOCUS displays the prompt:

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

3. Select the EXTERNAL INDEX option by entering:

```
EXTERNAL INDEX or 4
```

FOCUS responds:

```
NEW, OR ADD TO EXISTING INDEX DATABASE? (NEW/ADD)
```

Creating an External Index: The EXTERNAL INDEX Subcommand

4. For this example, assume you are creating a new index database and respond by entering:

NEW

FOCUS displays:

ENTER FILENAME OF EXTERNAL INDEX

5. Enter the name of the external index database:

EMPIDX

FOCUS then prompts you to:

ENTER NAME OF FOCUS/FUSION FILE TO INDEX

6. Enter the name of the database from which the index records are obtained:

EMPLOYEE

FOCUS displays:

ENTER NAME OF FIELD TO INDEX

7. Enter the name of the field:

CURR_JOBCODE

FOCUS displays:

ASSOCIATE INDEX WITH A PARTICULAR FIELD (YES/NO)

8. For this example, enter:

NO

FOCUS displays:

ANY RECORD SELECTION TESTS: (YES/NO)

9. For this example, enter:

NO

If you responded YES, FOCUS would have displayed the following:

ENTER IF OR WHERE TESTS (TERMINATE WITH 'END' ON A SEPARATE LINE)

Your record selection tests may span more than one line and must be terminated with the END command.

10. At this point, the REBUILD process continues and displays a screen similar to the following when it has completed:

```
EXTERNAL INDEX FILE:          EMPIDX
FULL NAME:                   EMPIDX.FOCUS
VERSION :
DATE/TIME OF LAST CHANGE:    05/13/99 15.40.46
```

```
EXTERNAL INDEX DATABASE PAGES: 00000001
DATABASE INDEXED:             EMPLOYEE
FIELD NAME:                   EMPINFO.CURR
FIELD FORMAT:                 A3
SEGMENT NAME:                 EMPINFO
SEGMENT LOCATION:             EMPLOYEE
```

EXTERNAL INDEX DATA COMPONENTS:

```
EMPLOYEE.FOCUS
EMP2.FOCUS
```

This output is from the ? FDT query, which is automatically performed at the end of the REBUILD EXTERNAL INDEX process. This is done in order to validate the contents of the index database.

Concatenating Index Databases

The external index feature enables indexed retrieval from concatenated FOCUS databases. If you wish to concatenate databases that comprise the index, you must issue the appropriate USE command prior to the REBUILD. The USE must include all cross-referenced and LOCATION files. REBUILD EXTERNAL INDEX contains an add function that enables you to append only new index records from a concatenated database to the index database, eliminating the need to recreate the index database.

The original database from which the index was built may not be in the USE list when you add index records. If it is, REBUILD EXTERNAL INDEX generates the following error message:

```
(FOC999) WARNING. EXTERNAL INDEX COMPONENT REUSED: ddname
```

Positioning Indexed Fields

The external index feature is useful for positioning retrieval of indexed values for defined fields within a particular segment in order to enhance retrieval performance. By entering at a lower segment within the hierarchy, data retrieved for the indexed field is affected, as the index field is associated with data outside its source segment. This enables the creation of a relationship between the source and target segments. The source segment is defined as the segment that contains the indexed field. The target segment is defined as any segment above or below the source segment within its path.

If the target segment is not within the same path, the following error message is generated:

```
(FOC974) EXTERNAL INDEX ERROR. INVALID TARGET SEGMENT
```

A defined field may not be positioned at a higher segment.

While the source segment can be a cross-referenced or location segment, the target segment cannot be a cross-referenced segment. If an attempt is made to place the target on a cross-referenced segment, the following error message is generated:

```
(FOC1000) INVALID USE OF CROSS REFERENCE FIELD
```

If you choose not to associate your index with a particular field, the source and target segments will be the same.

Activating an External Index

After building an external index database, you must associate it with the databases from which it was created. This is accomplished with the USE command. The syntax is the same as when USE is issued prior to building the external index database, except that the WITH or INDEX option is required.

Syntax **How to Activate an External Index**

The syntax is

```
USE  [ADD|REPLACE]
     database_name [AS mastername]

     index_database_name [WITH|INDEX] mastername
     .
     .
     .
END
```

where:

ADD

Appends one or more new databases to the present USE list. Without the ADD option, the existing USE list is cleared and then replaced by the current list of USE databases.

REPLACE

Replaces an existing *database_name* in the USE list.

database_name

Is the name of the database. In MVS, it is the ddname allocated to the database. In CMS, it is the file name, file type, and file mode.

You must include a database name in the USE list for all cross-referenced and LOCATION files that are specified in the Master File.

AS

Is used with a Master File name to concatenate databases.

mastername

Specifies the FOCUS Master File.

index_database_name

Is the name of the external index database. In MVS, this is the ddname allocated to the index database. In CMS, this is the file ID (file name, file type, file mode) of the index database.

WITH|INDEX

Is a keyword that creates the relationship between the component databases and the index database. INDEX is a synonym for WITH.

Reference **Special Considerations for External Indexes**

Consider the following when working with external indexes:

- Up to eight indexes can be activated at one time in a USE list using the WITH statement. More than eight indexes may be activated in a FOCUS session if you issue the USE CLEAR command and issue new USE statements.
- Up to 256 concatenated files may be indexed. However, only eight indexes may be activated at one time.
- Up to 240 partitions can be included in the creation of an external index, depending on whatever number of them is available in your File Control Table.
- Verification of the component files is now done for both the date and time stamp of file creation. Files with the same date and time stamp that are copied will yield the following messages:

```
(FOC995) ERROR. EXTERNAL INDEX DUPLICATE COMPONENT: fn  
REBUILD ABORTED
```

- MODIFY may only use the external index with the FIND or LOOKUP functions. The external index cannot be used as an entry point, such as:

```
MODIFY FILE filename.indexfld
```
- Indexes may not be created on field names longer than twelve characters.
- Text fields may not be used as indexed fields.
- The USE options NEW, READ, ON, LOCAL, and AS *master* ON *userid* READ are not supported for the external index database.
- The external index database need not be allocated, since CREATE FILE automatically does a temporary allocation. If a permanent database is required, then an allocation for the index database ddname must be in place prior to the REBUILD EXTERNAL INDEX command.
- SORTIN and SORTOUT, work files that the REBUILD EXTERNAL INDEX process creates, must be allocated with adequate space. In order to estimate the space needed, the following formula may be used:

```
bytes = (field_length + 20) * number_of_occurrences
```


Checking Database Integrity: The CHECK Subcommand

It is rare for the structural integrity of a FOCUS database to be damaged. Structural damage will occasionally occur, however, during a disk drive failure or if an incorrect Master File is used. In this situation, the REBUILD CHECK command performs two essential tasks:

- It checks pointers in the database.
- Should it encounter an error, it displays a message and attempts to branch around the offending segment or instance.

Although CHECK is able to report on a good deal of data that would otherwise be lost, it is important to remember that frequently backing up your FOCUS databases is the best method of preventing data loss.

CHECK will occasionally fail to uncover structural damage. If you have reason to believe that there is damage to your database, though CHECK reports otherwise, you can use a second method of checking database integrity. This method entails using the ? FILE and TABLEF commands. Though this is not a REBUILD function, it is included at the end of this section because of its relevancy to CHECK.

Procedure How to Use the CHECK Subcommand

To invoke REBUILD CHECK, follow these steps:

1. At the FOCUS prompt enter:

```
REBUILD
```

FOCUS prompts you:

```
Enter option
```

- | | |
|-------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

2. Select the CHECK option by entering:

```
CHECK or 5
```

FOCUS displays:

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

3. Enter the file ID of the database to be checked:

fileid

FOCUS immediately begins examining the database. If it does not find any errors, you will see the following message

```
NUMBER OF SEGMENTS RETRIEVED= n  
CHECK COMPLETED...
```

where:

n

Is the number of segment instances examined.

When FOCUS uncovers an error, it lets you know the type and location immediately using the following syntax

```
nnnn    SEGMENT    segname    type    PTR    pppp    aaaa
```

where:

nnnn

Is the number assigned to the segment. This is the same number that appears after invoking the CHECK command.

segname

Is the name of the segment.

type

Is the type of error:

DELETE indicates that the data has been deleted and should not have been retrieved.

OFFPAGE indicates that the address of the data is not on a page owned by this segment.

INVALID indicates that the type of linkage cannot be identified. It may be a destroyed portion of the database.

pppp

Is the block of data that contains the error.

aaaa

Is the offset of the error in hexadecimal.

Example Using the Check Option (File Undamaged) in CMS

```

rebuild
Enter option
  1. REBUILD           (Optimize the database structure)
  2. REORG            (Alter the database structure)
  3. INDEX            (Build/modify the database index)
  4. EXTERNAL INDEX   (Build/modify an external index database)
  5. CHECK            (Check the database structure)
  6. TIMESTAMP        (Change the database timestamp)
  7. DATE NEW         (Convert old date formats to smartdate formats)
  8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs
only)
  9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
5
CHECK
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
employee focus a
STARTING..
NUMBER OF SEGMENTS RETRIEVED=      576
CHECK COMPLETED...

```

Example Using the Check Option (File Damaged) in CMS

```

rebuild
Enter option
  1. REBUILD           (Optimize the database structure)
  2. REORG            (Alter the database structure)
  3. INDEX            (Build/modify the database index)
  4. EXTERNAL INDEX   (Build/modify an external index database)
  5. CHECK            (Check the database structure)
  6. TIMESTAMP        (Change the database timestamp)
  7. DATE NEW         (Convert old date formats to smartdate formats)
  8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
  9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
5
CHECK
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
rbbroken focus i

STARTING..

230 SEGMENT FUNDTRAN INVALID PTR   PAGE/ADDR 65536 272
242 SEGMENT FUNDTRAN INVALID PTR   PAGE/ADDR 28416 20
358 SEGMENT FUNDTRAN INVALID PTR   PAGE/ADDR 65536 272
404 SEGMENT FUNDTRAN OFFPAGE PTR   PAGE/ADDR 196608 272
491 SEGMENT FUNDTRAN INVALID PTR   PAGE/ADDR 62208 20

NUMBER OF SEGMENTS RETRIEVED=      574
CHECK COMPLETED...

```

Example Using the Check Option (File Undamaged) in MVS

```
rebuild
Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX             (Build/modify the database index)
 4. EXTERNAL INDEX    (Build/modify an external index database)
 5. CHECK             (Check the database structure)
 6. TIMESTAMP         (Change the database timestamp)
 7. DATE NEW          (Convert old date formats to smartdate formats)
 8. MDINDEX           (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE           (Convert FOCUS masters/DBs to FUSION)
> 5
CHECK
ENTER NAME OF FOCUS/FUSION FILE
>
> employee
STARTING..
NUMBER OF SEGMENTS RETRIEVED=      576
CHECK COMPLETED...
> >
```

Confirming Structural Integrity Using ? FILE and TABLEF

When you believe that there is damage to your database, though REBUILD CHECK reports there is not, use the ? FILE and TABLEF commands to compare the number of segment instances reported after invoking each command. A disparity indicates a structural problem.

Procedure How to Verify REBUILD CHECK Using ? FILE and TABLEF

1. At the FOCUS prompt, enter

```
? FILE filename
```

where:

```
filename
```

Is the name of the FOCUS database you are examining.

A report displays information on the status of the database. The number of instances for each segment is listed in the ACTIVE COUNT column.

2. To ensure that the TABLEF command in the next step counts all segment instances, including those in the short paths, issue the command:

```
SET ALL = ON
```

3. Enter

```
TABLEF FILE filename  
COUNT field1 field2  
END
```

where:

```
filename
```

Is the name of the Master File of the FOCUS database.

```
field1..
```

Are the names of fields in the database. Name one field from each segment; it does not matter which field is named in the segment.

The report produced shows the number of field occurrences for those fields named and thus the number of segment instances for each segment. These numbers should match their respective segment instance numbers shown in the ? FILE command (except for unique segments which the TABLEF command shows to have as many instances as are in the parent segment). If the numbers do not match, or if either the ? FILE command or TABLEF command ends abnormally, the database is probably damaged.

Example **Checking the Integrity of the EMPLOYEE Database**

User input is shown in lowercase; computer responses are in uppercase:

```
STATUS OF FOCUS FILE: EMPLOYEEFOCUS  A1 ON 05/13/1999 AT 16.17.32
          ACTIVE  DELETED      DATE OF    TIME OF    LAST TRANS
SEGNAME          COUNT    COUNT      LAST CHG   LAST CHG   NUMBER

EMPINFO          12
FUNDTRAN         6
PAYINFO          19
ADDRESS          21
SALINFO          70
DEDUCT           448
TOTAL SEGS       576
TOTAL CHAR       8984
TOTAL PAGES      8
LAST CHANGE                    05/13/1999  16.17.22  448

>
  set all = on
>
tablef file employee
count emp_id bank_name dat_inc type pay_date ded_code
end

PAGE          1

  EMP_ID  BANK_NAME  DAT_INC  TYPE  PAY_DATE  DED_CODE
  COUNT   COUNT     COUNT   COUNT  COUNT     COUNT
  -----  -----  -----  -----  -----  -----
         12         12         19     21         70         448

NUMBER OF RECORDS IN TABLE=          488 LINES= 1
```

Note that the BANK_NAME count in the TABLEF report is different than the number of FUNDTRAN instances reported by the ? FILE query. This is because FUNDTRAN is a unique segment and is always considered present as an extension of its parent.

Changing the Database Creation Date and Time: The **TIMESTAMP** Subcommand

FOCUS updates a FOCUS database's date and time stamp each time the database is changed by SCAN, FSCAN, CREATE, REBUILD, HLI, Maintain, or MODIFY. You can update a database's date and time stamp without making changes to the database by using REBUILD's **TIMESTAMP** subcommand.

Procedure How to Use the **TIMESTAMP** Subcommand

To invoke REBUILD **TIMESTAMP**, follow these steps:

1. At the FOCUS prompt, enter:

```
REBUILD
```

FOCUS prompts you:

```
Enter option
```

- | | |
|---------------------|--|
| 1. REBUILD | (Optimize the database structure) |
| 2. REORG | (Alter the database structure) |
| 3. INDEX | (Build/modify the database index) |
| 4. EXTERNAL INDEX | (Build/modify an external index database) |
| 5. CHECK | (Check the database structure) |
| 6. TIMESTAMP | (Change the database timestamp) |
| 7. DATE NEW | (Convert old date formats to smartdate formats) |
| 8. MDINDEX | (Build/modify a multidimensional index. FUSION DBs only) |
| 9. MIGRATE | (Convert FOCUS masters/DBs to FUSION) |

2. Select the **TIMESTAMP** option by entering:

```
TIMESTAMP or 6
```

FOCUS displays:

```
ENTER NAME OF FOCUS/FUSION FILE (FN FT FM)
```

3. Enter the file ID of the database whose date and time stamp is to be updated:

fileid

FOCUS prompts you for the source of the date and time:

ENTER OPTION: TODAY'S DATE (T), SEARCH FILE FOR DATE (D) OR MMDDYY
HHMMSS

where:

TODAY'S DATE (T)

Updates the database's date and time stamp with the current date and time.

SEARCH FILE FOR DATE (D)

Updates the database's date and time stamp with the last date and time at which the database was actually changed. FOCUS scans each page of the database and applies the most recent date and time recorded for a page to the database. This is the same as issuing the ? FILE query, and can be time consuming when the database is very large. This option is used to keep an external index database synchronized with its component database.

MMDDYY HHMMSS

Is a date and time that you specify, which REBUILD will use to update the database's date and time stamp. The date and time that you enter must have the format *mmddy hhmmss* or *mmddyymm hhmmss*. There must be a space between the date and the time. If you use two digits for the year, REBUILD uses the values for DEFCENT and YRTHRESH to determine the century.

4. Enter your selection (T, D, or a specific date and time).

If you supply an invalid date or time, FOCUS displays the following message:

(FOC961) INVALID DATE INPUT IN REBUILD TIME:

Converting Legacy Dates: The DATE NEW Subcommand

The REBUILD subcommand DATE NEW converts legacy dates (alphanumeric, integer, and packed-decimal fields with date display options) to smart dates (fields in date format) in your FOCUS databases. The utility uses update-in-place technology. It updates your database and creates a new Master File, yet does not change the structure or size of the database. You must back up the database before executing REBUILD with the DATE NEW subcommand. We recommend that you run the utility against the copy and then replace the original file with the updated backup.

Example Using the DATE NEW Subcommand in CMS

```
let clear *
set dfc = 19
set yrt = 50
use employee focus f
end
rebuild

Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX            (Build/modify the database index)
 4. EXTERNAL INDEX   (Build/modify an external index database)
 5. CHECK            (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)

date new

ENTER THE NAME OF THE MASTER
employee
ENTER THE NEW NAME FOR THE MASTER (FN FT FM)
newemp master a
HAVE YOU BACKED UP THE DATABASE? (YES,NO)
yes
```

If you answer anything other than YES to HAVE YOU BACKED UP THE DATABASE, REBUILD does not continue. Backing up your database is critical to this process.

In CMS, the new Master File is written to the file that you specified in the prompt. The default filetype is MASTER, and the default file mode is A.

Example Using the DATE NEW Subcommand in MVS

```
rebuild
Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG            (Alter the database structure)
 3. INDEX            (Build/modify the database index)
 4. EXTERNAL INDEX   (Build/modify an external index database)
 5. CHECK            (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index. FUSION DBs only)
 9. MIGRATE          (Convert FOCUS masters/DBs to FUSION)
> date new
DATE NEW
ENTER THE NAME OF THE MASTER
> employee
ENTER THE NEW NAME FOR THE MASTER
> newemp
HAVE YOU BACKED UP THE DATABASE? (YES,NO)
> yes
> NUMBER OF ERRORS=      0
NUMBER OF SEGMENTS=  11  ( REAL=    6  VIRTUAL=   5 )
NUMBER OF FIELDS=   34  INDEXES=   1  FILES=    3
TOTAL LENGTH OF ALL FIELDS= 365
HOLDING...
.
.
.
NUMBER OF SEGMENTS CHANGED=      107
```

In MVS, the new Master File is written to ddname HOLDMAST. After the new Master File is created, you should immediately copy it to a permanent data set.

For example:

```
DYNAM COPYDD HOLDMAST(NEWEMP) MASTER(NEWEMP)
```

How DATE NEW Converts Legacy Dates

REBUILD's DATE NEW subcommand overwrites the original legacy date field (an alphanumeric, integer, or packed-decimal field with date display options) with a smart date (a field in date format). When the storage size of the legacy date exceeds four bytes (the storage size of a smart date), a pad field is added to the database following the date field:

- Formats A6YMD, A6MDY, and A6DMY are changed to formats YMD, MDY, and DMY, respectively, and have a 2-byte pad field added to the Master File.
- The storage size of integer dates (I6YMD, I6MDY, for example) is 4 bytes, so no pad field is added.
- All packed fields and A8 dates add a 4-byte pad field.

When a date is a key field (but not the last key for the segment), and it requires a pad field, the number of keys in the SEGTYPE is increased by one for each date field that requires padding.

DATE NEW only changes legacy dates to smart dates. The field's format in the Master File must be one of the following (month translation edit options T and TR may be included in the format):

```
A8YYMD A8MDYY A8DMYY A6YMD A6MDY A6DMY A6YYM A6MYM A4YM A4MY
I8YYMD I8MDYY I8DMYY I6YMD I6MDY I6DMY I6YYM I6MYM I4YM I4MY
P8YYMD P8MDYY P8DMYY P6YMD P6MDY P6DMY P6YYM P6MYM P4YM P4MY
```

If you have a field that stores date values but does not have one of these formats, DATE NEW does not change it. If you have a field with one of these formats that you do not want changed, temporarily remove the date edit options from the format, run REBUILD DATE NEW, and then restore the edit options to the format.

Reference DATE NEW Usage Notes

- The DBA password for the database must be issued prior to issuing REBUILD.
- The original Master File cannot be encrypted.
- All files must be available locally during the REBUILD, including LOCATION files.
- The Master File cannot have GROUP fields.
- Some error numbers are available in &FOCERRNUM while all error numbers are available in &&FOCREBUILD. Test both &&FOCREBUILD and &FOCERRNUM for errors when writing procedures to rebuild your databases.
- To avoid any potential problems, clear all LETs and JOINs before issuing REBUILD.
- DEFCENT/YRTHRESH are respected at the global, database, and field level.
- Correct all invalid date values in the database before executing REBUILD/DATE NEW. The utility converts all invalid dates to zero. Invalid dates used as keys may lead to duplicate keys in the database.
- Adequate workspace, such as temporary attached disk storage, must be available for the temporary REBUILD file. As a rule of thumb, have space 10 to 20% larger than the size of the existing file available.
- REBUILD/INDEX is performed automatically if an index exists.
- REBUILD/REBUILD is performed automatically after REBUILD/DATE NEW when any key is a date.
- Sort libraries and work space must be available (as with REBUILD/INDEX). The REBUILD allocates default sort work space in MVS, if you have not already. DDNAMEs SORTIN and SORTOUT must be allocated prior to issuing a REBUILD.

What DATE NEW Does Not Convert

REBUILD's DATE NEW subcommand is a remediation tool for your FOCUS databases and date fields only. It does not remediate:

- DEFINE attributes in the Master File.
- ACCEPT attributes in the Master File.
- DBA restrictions (for example, VALUE restrictions) in the Master File or central security repository (DBAFIELD).
- Cross-references to other date fields in this or other Master Files.
- Any references to date fields in your FOCEXEC.

Using the New Master File Created by DATE NEW

REBUILD's DATE NEW subcommand creates an updated Master File that reflects the changes made to the database. Once the database has been rebuilt, the original Master File can no longer be used against the database. You must use the new Master File created by the DATE NEW subcommand.

The following procedure is an example of REBUILD DATE NEW in CMS:

```

USE
EMPLOYEE FOCUS F
END

REBUILD
DATE NEW
EMPLOYEE
NEWEMP
YES

-RUN
-IF (&&FOCREBUILD NE 0)OR(&FOCERRNUM NE 0) GOTO error_case;

USE
EMPLOYEE FOCUS F AS NEWEMP
END

TABLE FILE NEWEMP
PRINT ...
END

```

The new Master File is an updated copy of the original Master File except that:

- The USAGE format for legacy date fields is updated to remove the format and length. The date edit options are retained. For example, A6YMDTR becomes YMDTR.
- Padding fields are added for those dates that need them:

```
FIELDNAME= ,ALIAS= ,FORMAT=An,$ PAD FIELD ADDED BY REBUILD
```

where *n* is the padding length (either 2 or 4). Note that the FIELDNAME and ALIAS are blank.

- The SEGTYPE attribute is updated for segments that have remediated dates as keys when the date requires padding and the date is not the last field in the key. The SEGTYPE number will be increased by the number of pad fields added to the key.
- If the SEGTYPE is missing for any segment, the following line is added immediately prior to the \$ terminator for that segment:

```
SEGTYPE= segtype,$ OMITTED SEGTYPE ADDED BY REBUILD
```

where *segtype* is determined by FOCUS.

Converting Legacy Dates: The DATE NEW Subcommand

- If the USAGE attribute for any field—including date fields—is missing, the following line is added, immediately prior to the \$ terminator for that field:

```
USAGE= fmt, $ OMITTED USAGE ADDED BY REBUILD
```

where *fmt* is the format of the previous field in the Master File. FOCUS automatically assigns the previous field's format to any field coded without an explicit USAGE= statement.

Example Sample Master File: Before and After Conversion by DATE NEW

Before conversion	After conversion
FILE= <i>filename</i>	FILE= <i>filename</i>
SEGNAME= <i>segname</i> , SEGTYPE=S2	SEGNAME= <i>segname</i> , SEGTYPE=S3
FIELD=KEY1, , USAGE=A6YMD, \$	FIELD=KEY1, , USAGE= YMD, \$
	FIELD=, , USAGE=A2, \$ PAD FIELD ADDED BY REBUILD
FIELD=KEY2, , USAGE=I6MDY, \$	FIELD=KEY2, , USAGE= MDY, \$
FIELD=FIELD3, , USAGE=A8YYMD, \$	FIELD=FIELD3, , USAGE= YYMD, \$
	FIELD=, , USAGE=A4, \$ PAD FIELD ADDED BY REBUILD

When REBUILD's DATE NEW subcommand converts this Master File:

- The SEGTYPE changes from an S2 to S3 to incorporate a 2-byte pad field.
- Format A6YMD changes to smart date format YMD.
- A 2-byte pad field with a blank field name and alias is added to the Master File.
- Format I6MDY changes to smart date format MDY (no padding needed).
- Format A8YYMD changes to smart date format YYMD.
- A 4-byte pad field with a blank field name and alias is added to the Master File.

Action Taken on a Date Field During REBUILD/DATE NEW

A new message has been added after a REBUILD DATE NEW has been performed:

NUMBER OF SEGMENTS CHANGED= n

where:

n

Is the number of segments that have been changed. For example, if there are 10 fields on one segment, and 20 records, then n is 20 (the number of records/segments changed).

REBUILD/DATE NEW performs a REBUILD/REBUILD or REBUILD/INDEX automatically when a date field is a key or a date field is indexed. The following chart shows the action taken on a date field during the REBUILD/DATE NEW process.

Date Is a Key	Index	Result
No	None	NUMBER OF SEGMENTS CHANGED = n
No	Yes	REBUILD/INDEX on date field
Yes	None	REBUILD/REBUILD is performed.
Yes	On any field	REBUILD/REBUILD is performed. REBUILD/INDEX is performed for the indexed fields.

Migrating to a Fusion Database: The MIGRATE Subcommand

The MIGRATE subcommand is useful for migrating selected portions or an entire FOCUS database to Fusion. For a more detailed explanation of how to use this subcommand consult the *Fusion User's Manual for EDA 4.3 and FOCUS 7.1*.

Creating a Multi-Dimensional Index: The MDINDEX Subcommand

The MDINDEX subcommand is used to create or maintain a multi-dimensional index. For a more detailed explanation of how to use this subcommand consult the *Describing Data Manual*.

CHAPTER 12

Directly Editing FOCUS Databases With SCAN

Topics:

- Introduction
- Entering SCAN Mode
- Moving Through the Database and Locating Records
- Adding Segment Instances
- Moving Segment Instances
- Changing Field Contents
- Deleting Fields and Segments
- Saving Changes Made in SCAN Sessions
- Ending the Session
- Auxiliary SCAN Functions
- Subcommand Summary

SCAN is an interactive facility used for editing FOCUS databases. With it, you can edit FOCUS databases using subcommands similar to those used with text editors.

Introduction

SCAN permits you to:

- Add records to new or existing FOCUS databases.
- Change field values in FOCUS databases. With SCAN, you can change the values in KEY fields (not possible with MODIFY requests).
- Delete records from FOCUS databases.
- Search through FOCUS databases to locate instances of specified character strings or values.
- Display complete record contents showing all field values, or subsets of the fields in FOCUS databases.
- Move (relink) record segments and descendant segments from one parent record to another in FOCUS databases with parent-descendant structures.

In a typical SCAN session you identify a database and locate specific logical records of interest. Your knowledge of the database's structure and contents allows you to navigate from field to field. Within the database you can add or delete instances of data at the segment level or change data values at the field level.

Note: On databases protected with DBA passwords, SCAN is only available to those who have the proper password.

As you work in a SCAN session, your changes are accumulated in a revised version of your original database. When you decide to terminate your session, you can either save the changed version of the database and overwrite the original version with it, or keep the original version as it was when you started (if you have inadvertently changed the database).

We recommend that you copy your databases before using SCAN as an additional safety precaution; SCAN is a powerful tool for manipulating data, but keeps no log of the change transactions. Using the FOCUS Absolute File Integrity feature (SET SHADOW=ON) protects you against loss of data due to system crashes. (The SET SHADOW command is only effective if it has been issued prior to database creation. Consult the *Describing Data* manual for information about the Absolute File Integrity feature. See the *Developing Applications* manual for more information about SET parameters.)

SCAN vs. MODIFY, MAINTAIN, HLI, and FSCAN

FOCUS includes five facilities for maintaining the data in FOCUS databases. You should be aware of their differences:

- The SCAN facility is useful for examining the data in FOCUS databases to review or physically add, change, or delete data fields. With SCAN, an experienced user can quickly adjust database contents to correct errors or update fields. To use it effectively, however, you must know the database's contents and structure.

Caution: Because SCAN works directly on the data, there is the potential for corrupting data if you are unsure of the nature of your database. For example, if a SCAN operation such as REPLACE is issued against a database field such as SALES, without adequate selection criteria, every legitimate SALES field in the database could be overwritten by the replacement value, and all field values would have to be reentered.

- MODIFY (see Chapter 9, *Modifying Data Sources With MODIFY*) is a transaction processing environment that is used for maintaining FOCUS databases. MODIFY requests can be developed with elaborate match logic and data validation, as well as transaction logging. Such procedures, when fully tested, can be run by clerical personnel with no threat to database security.
- MAINTAIN, Information Builders' next generation data maintenance language, surpasses MODIFY, enabling you to write to FOCUS and non-FOCUS databases while providing support for record-at-a-time and set-based processing. MAINTAIN includes a new graphical user interface (the Winform Painter), and greatly enhances facilities for defining FOCUS transaction handling and cooperative processing operations. For more information about MAINTAIN, see Chapter 1, *Introduction to Maintain*.
- HLI (Host Language Interface) is an optional interface. It allows you to read and edit FOCUS databases from programs written in other programming languages (FORTRAN and C). HLI is similar to SCAN in function. HLI is described in the *Host Language Interface* manual.
- The FSCAN facility (see Chapter 13, *Directly Editing FOCUS Databases With FSCAN*) is similar to the SCAN facility: You can view, add, change, or delete data in your FOCUS databases. The FSCAN facility provides full-screen capabilities such as a prefix area and a command line. It also provides confirmation screens for DELETE and QUIT operations. Unlike SCAN, the FSCAN facility displays parent instances that lack descendant instances (short path records) and verifies acceptable test values defined with ACCEPT parameters. MARK and MOVE subcommands are not supported.

Entering SCAN Mode

From within FOCUS, enter SCAN mode by typing SCAN followed by FILE and the name of the FOCUS database to be scanned:

`SCAN FILE filename`

Moving Through the Database and Locating Records

After entering SCAN, your current position is at the top of the database. FOCUS databases are not sequential databases with one data record following another; they consist of segments. Databases can have one or more segments. The segments may have multiple instances of data (a Monthly Inventory segment holding a date and a quantity might have six instances in June and twelve in December). The collected data instances for a particular set of related segments constitute a logical record in the database.

The concept of a current line pointer (common in most system editors) is replaced in SCAN by the concept of a current position in the database, which represents a set of data instances that form a connected path within the database. Instead of processing databases line-by-line, SCAN achieves a somewhat similar effect by approaching FOCUS databases in a top-down, left-to-right scanning sequence.

As we said, on entering SCAN, you are automatically positioned at the top of the database. You may move through the entire database, or specify a subset of fields to be edited (called a Show List or a subtree). Show Lists are created with the SHOW subcommand, and they contain the fields you name (plus any intermediate segments required by FOCUS to navigate from one specified field to another). An important concept when specifying Show Lists is that the data in the selected records must meet all of the criteria specified in the SHOW subcommand.

What You See in SCAN Display Lines

When you display the contents of logical records in SCAN, each data field is identified on the screen by either its alias or the field name, whichever is shorter (and non-blank). Given the following Master File, the SCAN operation proceeds as shown below.

```

FILENAME=CAR, SUFFIX=FOC
SEGNAME=ORIGIN, SEGTYPE=S1
  FIELDNAME=COUNTRY, COUNTRY, A10, FIELDTYPE=I, $
SEGNAME=COMP, SEGTYPE=S1, PARENT=ORIGIN
  FIELDNAME=CAR, CARS, A16, $
SEGNAME=CARREC, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=MODEL, MODEL, A24, $
SEGNAME=BODY, SEGTYPE=S1, PARENT=CARREC
  FIELDNAME=BODYTYPE, TYPE, A12, $
  FIELDNAME=SEATS, SEAT, I3, $
  FIELDNAME=DEALER_COST, DCOST, D7, $
  FIELDNAME=RETAIL_COST, RCOST, D7, $
  FIELDNAME=SALES, UNITS, I6, $
SEGNAME=SPECS, SEGTYPE=U, PARENT=BODY
  FIELDNAME=LENGTH, LEN, D5, $
  FIELDNAME=WIDTH, WIDTH, D5, $
  FIELDNAME=HEIGHT, HEIGHT, D5, $
  FIELDNAME=WEIGHT, WEIGHT, D6, $
  FIELDNAME=WHEELBASE, BASE, D6.1, $
  FIELDNAME=FUEL_CAP, FUEL, D6.1, $
  FIELDNAME=BHP, POWER, D6, $
  FIELDNAME=RPM, RPM, I5, $
  FIELDNAME=MPG, MILES, D6, $
  FIELDNAME=ACCEL, SECONDS, D6, $
SEGNAME=WARENT, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=WARRANTY, WARR, A40, $
SEGNAME=EQUIP, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=STANDARD, EQUIP, A40, $

scan file car
SCAN:
next
COUNTRY=ENGLAND  CAR=JAGUAR  MODEL=V12X15E AUTO
TYPE=CONVERTIBLE SEAT= 4 DCOST= 7427 RCOST= 8878 UNITS= 0
LEN= 190  WIDTH= 66  HEIGHT= 48  WEIGHT= 3435  BASE= 105.0
FUEL= 18.0 BHP= 241 RPM= 5750 MPG= 16 ACCEL= 7

```

Note: SCAN uses ALIAS names instead of field names when aliases are shorter. Use DISPLAY (or CRTFORM) to display complete field names. Fields WARRANTY and STANDARD are not shown, because they do not lie on the path.

Identifying Data Fields in Scan

Some SCAN subcommands require that you specify particular data fields for the operation. LOCATE, for example, requires that you supply the data value for the target field. Within SCAN you can identify a data field in one of three ways:

- By its full field name as it appears in the Master File.
- By its alias.
- By the shortest unique truncation of either the field name or the alias.

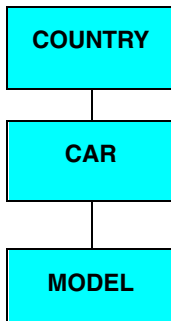
Ways to Move Through Databases

In SCAN sessions you can move from one segment instance directly to the next, jump from a parent segment instance to the first descendant field, or jump directly to a specific record of interest based on selection criteria specified in your request (for a description of these techniques, see *Subcommand Summary* on page 12-15).

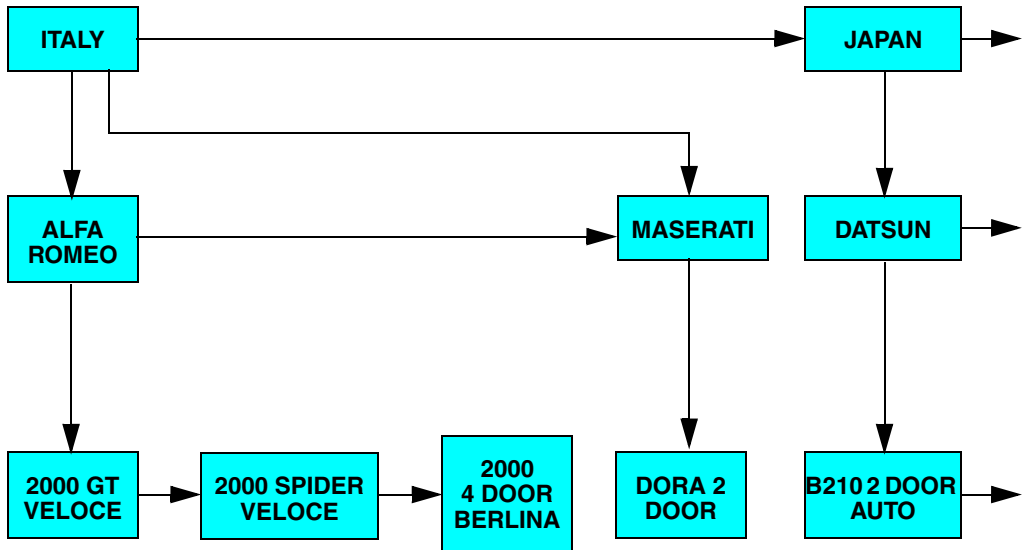
The examples in this section use the CAR database, mentioned in *What You See in SCAN Display Lines* on page 12-5. Enter SCAN, and then the subcommand:

```
SHOW COUNTRY CAR MODEL
```

This restricts the Show List to the first three segments of the database, as shown by this diagram:



The following schematic diagram shows how the data used in the examples is placed within the FOCUS structure:



There are six subcommands you may use to change the current position:

- TOP
- LOCATE
- TLOCATE
- NEXT
- JUMP
- UP

TOP

TOP moves the current position to the top of the database.

LOCATE

LOCATE moves the current position to the next record that fulfills certain conditions. Often, you use LOCATE to find a record with a certain value. For example, if your current position is near the top of the database and you enter the subcommand

```
LOCATE CAR=MASERATI
```

the following record appears:

```
COUNTRY = ITALY  CAR= MASERATI  
MODEL = DORA 2 DOOR
```

If you enter this subcommand again, SCAN searches for the next MASERATI record. Since there is only one MASERATI record, it moves the current position to the end of the database.

TLOCATE

TLOCATE moves the current position to the first record in the database that fulfills certain conditions. Often, you use TLOCATE to find a record with a certain value. For example, if you enter the subcommand

```
TLOCATE CAR=ALFA ROMEO
```

the following record appears regardless of where the current position was in the database:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

NEXT

The NEXT subcommand advances the current position to the next record. That is, it advances the current position one segment instance in the lowest segment in the Show List.

Suppose you entered SCAN to edit the CAR database and displayed the first record belonging to Italy by entering:

```
TLOCATE COUNTRY=ITALY
```

SCAN displays the following record:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

You then enter the subcommand NEXT:

```
NEXT
```

The lowest segment in this example is the MODEL segment. The MODEL instance in the record (2000 GT VELOCE) is the first of three instances descended from the car ALFA ROMEO. The NEXT subcommand moves the current position to the next instance in this chain, displaying the record:

```
ITALY  ALFA ROMEO  2000 SPIDER VELOCE
```


If you enter the NEXT subcommand again, SCAN displays:

```
ITALY ALFA ROMEO 2000 4 DOOR BERLINA
```

Now you are at the end of the MODEL under the instance ALFA ROMEO. If you enter the NEXT subcommand again, it moves the current position to the first MODEL chain under the next instance in the segment CAR. The next CAR instance is MASERATI. The record displayed is:

```
ITALY MASERATI DORA 2 DOOR
```

MASERATI has only one child instance, and it is the last car under the instance ITALY. If you enter the NEXT subcommand again, it moves the current position to the first MODEL chain under the next instance in the segment COUNTRY. The record displayed is:

```
JAPAN DATSUN B210 2 DOOR AUTO
```

JUMP

The JUMP subcommand moves the current position to the next segment instance in the segment you specify. The segment must have at least one field specified in the Show List.

Move the current position to the first record in the ITALY chain by entering:

```
TLOCATE COUNTRY=ITALY
```

This displays the record:

```
ITALY ALFA ROMEO 2000 GT VELOCE
```

Move the current position to the next car in the ITALY chain by entering:

```
JUMP CAR
```

Note: CAR is a field and not a segment name.

The following record appears:

```
ITALY MASERATI DORA 2 DOOR
```

Now return to the first record in the ITALY chain:

```
TLOCATE COUNTRY=ITALY
```

Jump to the next country in the database by entering:

```
JUMP COUNTRY
```

The following record appears:

```
JAPAN DATSUN B210 2 DOOR AUTO
```

UP

The UP subcommand moves the current position to the first instance in the lowest segment in the Show List descended from the segment that you specify.

Move the current position to the model 2000 SPIDER VELOCE:

```
TLOCATE MODEL=2000 SPIDER VELOCE
```

This displays the following record:

```
ITALY ALFA ROMEO SPIDER VELOCE
```

Move the current position to the first ALFA ROMEO model by entering:

```
UP CAR
```

The following records appears:

```
ITALY ALFA ROMEO 2000 GT VELOCE
```

Move the current position to the Maserati car:

```
LOCATE CAR=MASERATI
```

Move the current position to the first car in the ITALY chain by entering:

```
UP COUNTRY
```

The following record appears:

```
ITALY ALFA ROMEO 2000 GT VELOCE
```

Displaying Field Names and Field Contents

To view up to 64 fields, specify the SHOW subcommand. The SHOW subcommand does not list records lacking instances (short-path records).

To review field contents, use either the DISPLAY or TYPE subcommand.

TYPE Subcommand

At any point in a SCAN session, you may use the TYPE subcommand to display field names in a segment path (or those named in the SHOW subcommand, if one is in effect) and their contents for the current logical record (and/or several consecutive records).

DISPLAY Subcommand

DISPLAY produces a vertical list showing the full field names followed by the data values for the current logical record. DISPLAY allows you to select the fields to be displayed, and may include fields residing in segments picked up for the subtree but not actually named in the SHOW subcommand. This displays only the fields named in the SHOW subcommand if one is in effect.

Suppressing the Display

When moving through a database in SCAN with NEXT, JUMP, LOCATE, or TLOCATE, you automatically get a display of the contents of the next record unless you suppress the display. You do this by putting a period after the move keyword. Therefore,

```
NEXT .
```

retrieves, but does not display, the next record.

It is usually preferable to suppress the displays when performing global operations that affect many records.

Show Lists and Short-Path Records

If some segments lack data, it means that some logical records have missing segment instances. FOCUS discards short-path records when constructing the Show List.

Consider a subset of the CAR database. The subset has three segments with one field per segment (COUNTRY, CAR, MODEL). If you name all three fields in a SHOW subcommand, logical records that lack data in any of the specified fields are not selected for the subtree (they are short-path records).

The following example illustrates this. To run this example, enter the following commands as shown below. What you enter is in lowercase; computer responses are in uppercase.

```
scan file car
SCAN:
show country car
locate country=france
COUNTRY=FRANCE CAR=PEUGEOT
input car=renault
SCAN:
type
COUNTRY=FRANCE CAR=RENAULT
```

The example is as follows. The CAR database contains this data:

Country	Car	Model
.		
.		
.		
France	Peugeot	504 4 DOOR
France	Renault	
Italy	Alfa Romeo	2000 4 Door Berliner

Adding Segment Instances

Note that the French car Renault has no instances in the MODEL segment. A SCAN operation that names all three segments drops the logical record for Renault because Renault is missing instances in the MODEL segment, as follows.

```
show country car model
type 6
COUNTRY=ENGLAND CAR=JAGUAR MODEL=V12XKE AUTO
COUNTRY=ENGLAND CAR=JAGUAR MODEL=XJ12L AUTO
COUNTRY=ENGLAND CAR=JENSEN MODEL=INTERCEPTOR III
COUNTRY=ENGLAND CAR=TRIUMPH MODEL=TR7
COUNTRY=FRANCE CAR=PEUGEOT MODEL=504 4 DOOR
COUNTRY=ITALY CAR=ALFA ROMEO MODEL=2000 4 DOOR BERLINER
```

Note: In all of the examples in this section, user input is shown in lowercase; the FOCUS response is in uppercase.

To locate short-path records that will be dropped from a Show List, make a test pass through the database at the short-path level to see what is there before issuing the Show List for the edit operation. (This is highly recommended when adding new records to a database.) Thus, for the simple previous example, if you start by making a pass through the database selecting all records containing values for COUNTRY and CAR, you will find the Renault car.

```
show country car
type 6
COUNTRY=ENGLAND CAR=JAGUAR
COUNTRY=ENGLAND CAR=JENSEN
COUNTRY=ENGLAND CAR=TRIUMPH
COUNTRY=FRANCE CAR=PEUGEOT
COUNTRY=FRANCE CAR=RENAULT
COUNTRY=ITALY CAR=ALFA ROMEO
```

On the next pass, you add the MODEL segment and note that Renault disappears (due to the short-path). Knowing this, you refrain from adding a potential duplicate record for France and make a mental note to make another pass to update the short-path record with data for the MODEL segment.

Adding Segment Instances

The INPUT subcommand is used to add new segment instances to the database. New segment instances are inserted into the database in the correct sort sequence, as long as you have avoided adding duplicate instances to existing segments. Duplicate instances may not be found if they lack field values (short-path records). See *INPUT Command* on page 12-24 for a description of the syntax and an example of its use.

Moving Segment Instances

Use the MOVE subcommand to move a segment instance and all of its descendants from one parent segment to another. The operation requires several steps:

1. Locate the record to be moved and mark it with the MARK statement (see *MARK Command* on page 12-27).
2. Move the current position to the new parent record (see *LOCATE Command* on page 12-26 and *TLOCATE Command* on page 12-35).
3. Issue the MOVE subcommand indicating the field name that identifies the segment instance to be moved.

MOVE Command on page 12-28 describes how the segment is integrated into the database structure.

Changing Field Contents

CHANGE and REPLACE alter the contents of data fields.

Use CHANGE to substitute one character string for another, and REPLACE to substitute a new value for a field. CHANGE is issued to change alphanumeric strings within fields. REPLACE is used with either alphanumeric or numeric fields to replace the entire contents of the field(s).

Both operations can be applied to one or more instances from the current position to the end of the database. To change all instances in the database, use TLOCATE to find the first record before entering the CHANGE or REPLACE subcommand.

See *CHANGE Command* on page 12-18 and *REPLACE Command* on page 12-31 for additional information about CHANGE and REPLACE.

Deleting Fields and Segments

DELETE removes one or more instances of data in one or more segments containing the named field (and all descendant segment instances).

Saving Changes Made in SCAN Sessions

The SAVE subcommand writes all pending changes to the FOCUS database and leaves you in SCAN mode. Most installations recommend that SAVE operations be performed periodically to protect against accidental loss of update results due to communications failure or other processing interruptions.

Ending the Session

When ending the SCAN session, you can exit with or without saving your changes.

Exiting and Saving the Changes

To end the SCAN session, write the changes to the FOCUS database, and return to the FOCUS command level; use either the FILE subcommand or its synonym, END.

Exiting Without Saving the Changes

To leave SCAN and return to FOCUS without writing pending changes to the FOCUS database, use the QUIT subcommand.

Caution: The use of this subcommand does not guarantee that all changes to the database will be ignored. During SCAN execution, large buffer areas hold database records. Depending on the operating system in use and the size of these buffer areas, it is possible that a large SCAN change file could threaten the capacity of the temporary buffer storage, in which case the operating system might write the pending changes to the database to clear the buffer. This would update your database.

Auxiliary SCAN Functions

SCAN provides two convenience features: the first displays or executes a previous command; the second substitutes a one-character value for a complete SCAN subcommand.

Displaying a Previous SCAN Subcommand

To display the last subcommand issued, use the ? subcommand.

To re-execute the previous subcommand, use the AGAIN subcommand. This is particularly useful when finding multiple instances of a field value with LOCATE.

Preset X or Y to Execute a SCAN Subcommand

To set X (or Y) equal to another SCAN subcommand, type the syntax

```
{X|Y} subcommand
```

where:

```
subcommand
```

Is a SCAN subcommand.

This gives you an alias for a long, frequently-used subcommand. For example, to substitute Y for a DISPLAY subcommand showing the first and last names of the employee at the current position in the database, type:

```
Y DISPLAY FN LN
```

The next time you type Y and press the Enter key, this DISPLAY subcommand is issued.

Subcommand Summary

SCAN subcommands can be entered as unique truncations or in full. In the summary below, the capital letters represent the shortest unique truncations. A list of descriptions of these subcommands, with additional information and examples, begins with *AGAIN Command* on page 12-16.

Subcommand	Function
Again	Repeat the last subcommand.
BAck	Go back to a previously marked logical record (see MArk, below).
CHAnge	Change a character string.
CRTform	Display a list of fields on a CRTFORM.
DElete	Delete one or more instances of the segment containing the named field (and all descendant segments).
DIisplay	Display the data values for the fields specified.
End	Terminate the SCAN session and write the changes to the database.
File	Terminate the SCAN session and write the changes to the database.
Input	Enter a new record.
Jump	Jump to the next or nth occurrence of field.

Subcommand Summary

Subcommand	Function
Locate	Search for records that match the selection criteria.
MArk	Mark a record so that you can return to it later in the SCAN session.
MOve	Relink the segment to another parent.
Next	Move <i>n</i> records ahead.
Quit	End the session and drop the pending changes.
Replace	Replace a field value in one or more instances.
SAve	Save all pending changes and continue.
SHow	Select a subset of the fields in the database (a logical view—Show List).
TLocate	Go to top of database, then locate record(s) meeting the selection criteria.
TOp	Reset current position at first logical record in the database.
TYpe	Type record(s).
UP	Move current position to parent segment's first descendant.
X	Used for command substitution.
Y	Same as X above.
?	Print the previous subcommand.

AGAIN Command

The AGAIN command tells the system to repeat the previous valid command.

This is particularly useful after LOCATE, as it continues the search for the next instance of the target value.

Syntax **AGAIN Command**

Again

Example Using the AGAIN Command

```
show emp_id last_name salary dpt
locate dpt=mis
  EID=112847612 LN=SMITH  DPT=MIS   SAL= 13200.00
again
  EID=117593129 LN=JONES  DPT=MIS   SAL= 18480.00
```

LOCATE retrieves the first record following the current position that matches the test condition. AGAIN repeats the process, as if the LOCATE statement had been retyped, and the next record that meets the test condition is displayed.

The fields displayed above are those named in the previous SHOW subcommand. The DPT (Department) field is available in the Show List because it resides in the same segment as the EMP_ID and LAST_NAME fields.

Reference Commands Similar to Again

Within SCAN, entering a question mark (?) causes a display of the last subcommand to be executed. If you wish to execute it again, reenter the command or use AGAIN.

BACK Command

The BACK subcommand works in conjunction with the previous MARK subcommand (only one MARK is in effect at a time). When BACK is issued, control returns to the previous marked record (see MARK subcommand).

Syntax BACK Command

```
BAck
```

Example Using the BACK Command

```
show emp_id last_name first_name salary
next
  EID=071382660 LN=STEVENS FN=ALFRED  SAL= 11000.00
jump emp_id 2
  EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
mark
next 2
  EID=119265415 LN=SMITH   FN=RICHARD SAL= 9500.00
back
  EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
```

Reference Commands Similar to BACK

None.

CHANGE Command

CHANGE is used to replace specified alphanumeric character strings with new strings in data fields. Changes may be made sequentially to every record in the database, or to all records that match a LOCATE criteria.

Note: CHANGE cannot be used on numeric fields with formats I,P,F, and D.

Syntax

CHANGE Command

```
CHAnge field=/oldstring/newstring/, $ [*|n]
```

A period (.), colon (:), or slash (/) may be used as the string delimiter and must be the first character after the equal sign (=). The same character must then be used to terminate the old and new strings.

The replication factor n (where n is number of strings to be replaced) has a default value of 1. When more than one string is to be changed, indicate the replication factor as a single digit following the line terminator characters, \$. To replace all instances of the string in the remainder of the database, use the asterisk (*). To replace all instances of the string in the database, issue TOP before the CHANGE. This resets the current position at the first logical record.

Example

Single-Field Change With the CHANGE Command

To change a single field, first locate it then make the change.

```
show emp_id last_name first_name
tlocate ln=stevens
EID=071382660 LN=STEVENS FN=ALFRED
change ln=/stevens/stephens/, $
EID=071382660 LN=STEPHENS FN=ALFRED
```

Example Sequential Changes With the CHANGE Command

To change all occurrences of the old string to the new string throughout the database starting at the current position, use the replication factor, *.

```
show last_name department salary
locate dpt=mis
LN=SMITH      DPT=MIS  SAL= 13200.00
change dpt=/mis/mis dept/, $ *
LN=SMITH      DPT=MIS  DEPT  SAL= 13200.00
LN=JONES      DPT=MIS  DEPT  SAL= 18480.00
LN=JONES      DPT=MIS  DEPT  SAL= 17750.00
LN=MCCOY      DPT=MIS  DEPT  SAL= 18480.00
LN=BLACKWOOD  DPT=MIS  DEPT  SAL= 21780.00
LN=GREENSPAN  DPT=MIS  DEPT  SAL= 9000.00
LN=GREENSPAN  DPT=MIS  DEPT  SAL= 8650.00
LN=CROSS      DPT=MIS  DEPT  SAL= 27062.00
LN=CROSS      DPT=MIS  DEPT  SAL= 25775.00
VALUES REPLACED= 6
EOF:
```

The VALUE REPLACED parameter displayed at the bottom of the report shows how many segment instances were changed, not how many lines SCAN displays after the change.

Example Match Logic Changes With the CHANGE Command

The current position is reached through a LOCATE (or TLOCATE) subcommand, and the conditions of the LOCATE are retained and applied in selecting records to be changed.

```
tlocate dpt=mis dept, sal lt 15000
LN=SMITH      DPT=MIS  DEPT  SAL=13200.00
change dpt=/mis dept/mis/ , $ *
LN=SMITH      DPT=MIS           SAL= 13200.00
LN=GREENSPAN  DPT=MIS           SAL= 9000.00
LN=GREENSPAN  DPT=MIS           SAL= 8650.00
VALUES REPLACED= 2
EOF:
```

Here SCAN changes only two segment instances rather than the six instances in the previous example, but three are shown because there are two child segments for the GREENSPAN record.

Note:

- If CHANGE is immediately preceded by LOCATE or TLOCATE, only the instances that satisfy the LOCATE conditions are changed.

Subcommand Summary

- If no record selection criteria is included, the CHANGE action will change subsequent instances. Changed field instances may include descendant instances not represented in the Show List.

Reference **Commands Similar to CHANGE**

REPLACE is used to replace the entire contents of numeric or alphanumeric fields.

CRTFORM Command

The CRTFORM subcommand formats the display of selected data fields. Enter the field names separated by blanks. (The selection begins at the current position.) The display aligns two fields per line where possible.

Use the TYPE subcommand to display the results of a CRTFORM subcommand.

Syntax **CRTFORM Command**

```
CRTform * {*|fieldname [*]...fieldname}
```

You can enter the full field names, aliases, or the shortest unique truncations of either. To display all fields between two named fields, place an asterisk in the list of field names. To simply display all fields, use an asterisk in place of the field names.

Example **Specifying Individual Fields With CRTFORM**

```
crtform eid ln fn sal  
type
```

```
EMP_ID          =071382660      LAST_NAME       =STEVENS  
FIRST_NAME      =ALFRED          SALARY          = 11000.00
```

Example **Specifying All Fields Between Two Named Fields With CRTFORM**

```
crtform eid * salary  
type
```

```
EMP_ID          =071382660      LAST_NAME       = STEVENS  
FIRST_NAME      =ALFRED          HIRE_DATE       = 800602  
DEPARTMENT      =PRODUCTION     CURR_SAL        = 11000.00  
CURR_JOBCODE    =A07            ED_HRS          = 25.00  
BANK_NAME       =                BANK_CODE       =  
BANK_ACCT       =                EFFECT_DATE     = 0  
DAT_INC         =820101         PCT_INC         = .10  
SALARY          =11000.00
```

Reference **Commands Similar to CRTFORM**

None.

DELETE Command

The segment containing the field name is deleted and all of its descendant segments are deleted. Any references to indexed fields are removed from their associated indexes.

Note:

- If DELETE is immediately preceded by a LOCATE subcommand, then only instances that satisfy the LOCATE conditions are deleted.
- If no record selection criteria is included, the delete action will remove subsequent instances. Deleted field instances may include descendant segments that are not represented in the Show List.

None of the changes made during a SCAN session take effect until you save them. When you do write them to the database using SAVE or FILE (see descriptions of these subcommands on the following pages), they become permanent; thus you should closely monitor the effect of your changes as you work in SCAN. If you make a mistake, it is important to QUIT immediately to avoid any permanent damage.

Syntax **DELETE Command**

```
DElete fieldname [factor]
```

where:

factor

Is one of the following:

1 is the default value.

* deletes all instances of the field.

n is the number of data instances to be deleted. When more than one instance is to be deleted, indicate the replication factor as a numeric value following the line terminator characters ,\$.

Example **Using DELETE**

```
show emp_id last_name salary jobcode
next
EID=071382660 LN=STEVENS SAL= 11000.00 JBC=A07
delete jobcode 6
SEGMENTS DELETED= 6
```

The next six instances of JOBCODE are removed.

Reference **Commands Similar to DELETE**

None.

DISPLAY Command

This subcommand displays the values of the named fields in a neat vertical list, whether the field is in the SHOW list or not. It is useful to view the values of fields not specified in a SHOW list. (TYPE presents only the fields named in the SHOW command.) It is convenient, for example, to move through databases looking at only the values of a few key fields. Then, when you find the record you want, use DISPLAY to display all of the fields in the segment(s) contained in the Show List.

The DISPLAY subcommand does not remain in effect. It simply lists the specified values. If you need to issue it repeatedly, store it with the X or Y subcommand for subsequent execution.

Syntax **DISPLAY Command**

```
DISplay fieldname [fieldname...fieldname]
```

The field identifier may be the full field name, alternate alias, or shortest unique truncation of either. Separate field names from each other with spaces.

Example **Using DISPLAY**

```
show last_name dat_inc
locate ln =smith
    LN=SMITH      DI=820101
display last_name first_name salary department
LAST_NAME      =SMITH
FIRST_NAME     =MARY
SALARY         = 13200.00
DEPARTMENT     =MIS
```

If the DISPLAY subcommand does not produce a list, it indicates that the fields requested must lie outside the currently retrieved segment(s) by displaying the message:

```
NO CURRENT VALUE FOR: field.
```

Reference **Commands Similar to DISPLAY**

- The TYPE subcommand is also used for showing the contents of the currently active data fields. TYPE presents the data horizontally, using the shortest name or alias available in the Master File. DISPLAY presents the information vertically, showing the full field names.
- CRTFORM is used to format a screen, showing the full field names and the field contents, blocked two to a line. Use TYPE to show the contents of the CRTFORM.

END Command

Terminates the SCAN session and writes all pending modifications to the FOCUS database.

Syntax **END Command**

End

Example **Using the END Command**

END

Reference **Commands Similar to END**

- The FILE subcommand is a synonym for END. This also results in normal termination of the session.
- The SAVE subcommand also writes the modifications to the database, but does not terminate the SCAN session. You retain your position in the database.

FILE Command

Terminates the SCAN session and writes all pending modifications to the FOCUS database.

Syntax **FILE Command**

File

Example **Using the FILE Command**

FILE

Reference **Commands Similar to FILE**

- The END subcommand is a synonym for FILE. This also results in normal termination of the session.
- The SAVE subcommand writes the modifications to the database, but does not terminate the SCAN session. You retain your position in the database.

INPUT Command

The subcommand opens the database to accept one or more new segments of data. It creates a segment instance in each segment for which a field value is specified.

The new records are inserted after the record currently displayed; that is, they break the chain. However, if the segment is being maintained in some sort sequence, a check is subsequently performed and the new records inserted in their proper positions.

Syntax INPUT Command

```
Input [field=value,...[,,$]]
```

The input records are defined as free-format, or comma-delimited. They are entered in one of two ways:

- The data may be typed on the same line as the command. It must be typed on one line. In this case, it does not have to be terminated by a comma and dollar sign (,\$).
- Or if the subcommand is issued on a line by itself, then the new record may be typed on several lines, but it *must* be terminated by a comma and dollar sign (,\$).

Example Using the INPUT Command

```
show emp_id last_name salary jobcode
tlocate ln=jones
EID=117593129 LN=JONES SAL= 18480.00 JBC=B03
input salary=19000.00, jobcode=b04
SCAN:
type
EID=117593129 LN=JONES SAL= 19000.00 JBC=B04
```

Caution: SCAN rejects records that have key field values that already exist in the database (duplicate keys). In this example, if you type the following, you get a warning.

```
input eid=117593129, salary=19000.00, jobcode=604
DATA KEYS ARE ALREADY IN FILE
SCAN:
```

Such warnings are only provided for *key* fields, however, and inadvertently creating a duplicate instance of a segment can have unexpected consequences, particularly if one of the records is a short-path record. Subsequently, you may see different versions depending on the fields you name in your SHOW command.

Reference Commands Similar to INPUT

None.

JUMP Command

Starting from the field in the current record, JUMP moves immediately to the next occurrence of the same field. This skips over any intervening records and is a quick way to traverse a database. Specify *n* to jump *n* occurrences.

If JUMP encounters no additional field occurrences for the same parent record, it stops at the last record in the current chain and displays the END-OF-CHAIN message. It does not move to the start of the next chain.

Syntax **JUMP Command**

```
Jump fieldname [n]
```

Example **Using the JUMP Command**

```
show emp_id last_name first_name salary
type 7
EID=071382660 LN=STEVENS FN=ALFRED SAL= 11000.00
EID=071382660 LN=STEVENS FN=ALFRED SAL= 10000.00
EID=112847612 LN=SMITH FN=MARY SAL= 13200.00
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
EID=117593129 LN=JONES FN=DIANE SAL= 17750.00
EID=119265415 LN=SMITH FN=RICHARD SAL= 9500.00
EID=818692173 LN=CROSS FN=BARBARA SAL= 25775.00
EID=119265415 LN=SMITH FN=RICHARD SAL= 9050.00

top
TOF:
next
EID=071382660 LN=STEVENS FN=ALFRED SAL= 11000.00
jump emp_id 2
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
```

Reference **Commands Similar to JUMP**

The NEXT subcommand is used to advance to the next logical record.

LOCATE Command

Starting at the current position, initiates a search for record(s) meeting the test condition(s). When an acceptable record is found, it is displayed. If the end of the database is encountered during the search, the message EOF: is displayed.

Syntax LOCATE Command

```
Locate field rel value [[AND|,]field rel value [,,$] [*|n]]
```

where:

field

Is the field name of the target(s).

rel

Is one of the following test relations:

Relation	Meaning
EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS	Contains
OMITS	Omits

value

Is the object of the comparison.

n

Is the number of occurrences which may exist.

The comma-dollar sign (,\$) terminator symbol is not required if only one record is sought (the default). It is required if you provide a replication factor (*n*) larger than 1. If the replication factor is set to *, then all records meeting the test conditions are displayed (from the current position to the end of the database).

When using more than one test relation, separate them by either commas or the word AND, as

```
locate field rel value, field rel value
```

or:

```
locate field rel value AND field rel value
```

If you supply a list of values with an EQ test, separate the values with the word OR:

```
locate field EQ value OR value OR value
```

Example Using the LOCATE Command

```
show emp_id last_name first_name salary
locate dpt=mis
EID=112847612 LN=SMITH SAL= 13200.00 JBC=B14
```

Reference Commands Similar to LOCATE

TLOCATE has exactly the same function, but effectively adds the TOP function and begins the search at the top of the database.

MARK Command

The MARK subcommand identifies a logical record so that you can return to it when you issue the MOVE or BACK subcommand. Only one record can be marked at a time. MARK is used to identify data to be moved to a new location in the database, and to return to a record with the BACK command.

Syntax MARK Command

```
MArk
```

Example Using the MARK Command

```
show emp_id last_name first_name salary
next
EID=071382660 LN=STEVENS FN=ALFRED SAL= 11000.00
jump emp_id 2
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
mark
next 2
EID=119265415 LN=SMITH FN=RICHARD SAL= 9500.00
back
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
```

Reference Commands Similar to MARK

None.

MOVE Command

The MOVE subcommand moves segment instances and all of their descendant segments from one parent segment to another.

Identify the record instance of the segment to be moved with the MARK subcommand. Then locate the new position for the marked segment instance in any manner (LOCATE, NEXT, etc.). Follow with the MOVE subcommand naming the instance of the segment being moved. The moved instance and all of its descendants are made descendants of the parent at the current position. If the SEGTYPE is not S or SH, then the segment will be inserted after the record currently shown. If the SEGTYPE is S or SH (sorted, sorted high-to-low), the segments will be located in the proper sort sequence.

Syntax **MOVE Command**

MOVe fieldname

Example **Using the MOVE Command**

```
show emp_id last_name salary dat_inc
next
  EID=071382660 LN=STEVENS    DI=820101 SAL= 11000.00
mark
locate ln=greenspan
  EID=543729165 LN=GREENSPAN DI=820611 SAL= 9000.00
move dat_inc
  EID=543729165 LN=GREENSPAN DI=820101 SAL=11000.00
```

In the example, the date of increase (DAT_INC or DI) and salary (SAL) are taken from the marked record of Alfred Stevens and moved to Mary Greenspan's record.

Reference **Commands Similar to MOVE**

None.

NEXT Command

The current position is advanced *nn* records and the new position is displayed (where *nn* is the number of records from 1 to 99). If the end of the database is reached during the movement to the new current position, the message EOF: is displayed.

Syntax **NEXT Command**

Next [*nn*]

The default is one record.

Example **Using the NEXT Command**

```
show emp_id last_name first_name salary
type 8
EID=071382660 LN=STEVENS FN=ALFRED SAL= 11000.00
EID=071382660 LN=STEVENS FN=ALFRED SAL= 10000.00
EID=112847612 LN=SMITH FN=MARY SAL= 13200.00
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
EID=117593129 LN=JONES FN=DIANE SAL= 17750.00
EID=119265415 LN=SMITH FN=RICHARD SAL= 9500.00
EID=119265415 LN=SMITH FN=RICHARD SAL= 9050.00
EID=119329144 LN=BANNING FN=JOHN SAL= 29700.00
```

top

TOF:

next 4

```
EID=117593129 LN=JONES FN=DIANE SAL= 18480.00
```

next 2

```
EID=119265415 LN=SMITH FN=RICHARD SAL= 9500.00
```

NEXT 4 advances the current position to the fourth logical record and displays the field values at that position. The subsequent NEXT 2 moves the current position forward two more logical records.

Reference **Commands Similar to NEXT**

None.

QUIT Command

Ends the SCAN session. All pending modifications to the database (those not yet written permanently to the disk) are suppressed.

The use of this subcommand *does not guarantee* that all changes to the database will be ignored. During SCAN execution, large buffer areas hold the pending changes. Depending on the operating system and buffer sizes, a large SCAN file could threaten the buffer capacity. This forces the operating system to write your pending changes to the database to clear the buffer. This would update your database, even though you had not issued a SAVE, END or FILE subcommand.

The FOCUS Absolute File Integrity facility reduces the risk of making changes you do not want. Also, keeping your own copy of the database before you start the session gives you a recovery capability in the event you lose your way in SCAN and create a database you subsequently decide to discard.

The QUIT subcommand acts only to prevent transfer of those records in the buffer to the disk.

When a change is made to a database immediately prior to issuing QUIT, the change is usually suppressed. If SCAN activity is high between modifications to the database, however, the chance of suppressing all changes is less likely, because the buffer work areas may, of necessity, have been written to the disk to make way for more pages of database records.

Syntax **QUIT Command**

Quit

Example **Using the QUIT Command**

QUIT

Reference **Commands Similar to QUIT**

END and FILE both terminate the session but both write any pending changes to the database. SAVE also writes the changes to the database, but leaves you in the SCAN session.

REPLACE Command

The REPLACE command replaces the data values for the record at the current position with the data values provided. The fields replaced may reside on the same segment or different segments, but must be on the path defined by the Show List if one is in effect.

Two types of global REPLACE operations can be specified:

- **Sequential replacement:** If the current position was not reached using a prior LOCATE subcommand, the replication factor applies to this record and the next $n-1$ records retrieved.
- **Matched replacement:** If a prior LOCATE subcommand established the current position, the search criteria remains in effect and the replication factor applies to this logical record and the next $n-1$ records that also meet the search criteria.

Syntax

REPLACE Command

```
Replace [KEY] field=value, field=value, $ [factor]
```

where:

factor

Is one of the following:

1 is the default.

* represents all fields.

nn is the number of field values that can be replaced at one time. If the replication factor *nn* is greater than 1, then all of the replaced fields must reside on the same segment.

If the field whose value is being replaced is used to keep the segment in the proper sort sequence (that is, it is a key field), then the word KEY must be placed after the command. Without this word, a message is displayed indicating that the key field cannot be replaced.

Note: The replication factor cannot be used with REPLACE KEY.

Example

Replacing a Field Value With REPLACE

```
show emp_id last_name salary
tlocate eid=112847612
  EID=112847612 LN=SMITH SAL= 13200.00
replace salary=16000.00
  EID=112847612 LN=SMITH SAL= 16000.00
```

Example Replacing Multiple Field Values With REPLACE

```
show emp_id last_name jobcode
next
  EID=071382660 LN=STEVENS JBC=A07
replace jobcode=B02,$ *
  EID=071382660 LN=STEVENS JBC=B02
  EID=071382660 LN=STEVENS JBC=B02
  EID=112847612 LN=SMITH JBC=B02
.
.
.
VALUES REPLACED= 19
EOF:
```

Example Replacing a Key Field Value With REPLACE

```
show emp_id last_name first_name
tlocate ln=stevens
  EID=071382660 LN=STEVENS FN=ALFRED
replace key eid=971382660
  EID=971382660 LN=STEVENS FN=ALFRED
KEY VALUE RESEQUENCED...
type *
  EID=971382660 LN=STEVENS FN=ALFRED
EOF:
```

Notes on replacing key fields:

- The segment is re-sequenced to preserve the correct sort order. In this case, we gave Stevens the highest employee number in the database, so the TYPE * command types one record and reaches end-of-file.
- Only one key field can be replaced at a time.
- This may result in duplicate keys in the database (you need to keep track of this).

Reference Commands Similar to REPLACE

CHANGE command.

SAVE Command

Writes out all modifications to the FOCUS database. The SCAN session continues at the current position held before the SAVE. If the FOCUS Absolute File Integrity feature is active, this is the point at which a new checkpoint is taken.

To activate the Absolute File Integrity feature, issue the following command at the FOCUS command level before you create the database:

```
SET SHADOW=ON
```

If the SET SHADOW command is issued after the database is created, the command has no effect. See the *Describing Data* manual for information about the FOCUS Absolute File Integrity feature. See the *Developing Applications* manual for more information about the SET parameters.

Periodic use of SAVE during SCAN sessions is recommended. Otherwise, if communication lines are lost or other processing interruptions occur, the modifications made since the previous SAVE must be repeated.

Syntax **SAVE Command**

```
SAve
```

Example **Using the SAVE Command**

```
SAVE
```

All modifications to the database are written to the disk, and the SCAN session continues.

Reference **Commands Similar to SAVE**

Both END and FILE write your changes to the database and terminate the SCAN session. QUIT is used to delete any pending changes to the database and terminate the SCAN session.

SHOW Command

SHOW is used to create a subset of the database (called a Show List, subtree, or a logical view) for editing. It always moves the current position to the top of the database, and the logical records are only as deep as the Show List (that is, they consist of only the segments named in the SHOW subcommand, which had data in all of the specified fields plus any intermediate segments needed to connect the segments containing the named fields).

Syntax **SHOW Command**

`SHow [fieldlist]`

where:

`fieldlist`

Can be one of the following:

```
fieldname [*] fieldname
* fieldname
fieldname *
```

Separate field names with blanks. Field names can be full field names, aliases, or unique truncations of either.

On entry into the SCAN environment, all of the data fields in the first physical top-to-bottom path are displayed as the default Show List. When SHOW is issued with no list of field names, the names of all of the fields in the current path are displayed.

Use an asterisk (*) between two field names to select all fields between and including them. Use an asterisk and one field name to select all field names up to and including the named field. Use one field name and an asterisk to select all field names from that field on.

Example **Selecting a Logical View (a Show List)**

```
show eid last_name salary
type *
EID=071382660 LN=STEVENS SAL= 11000.00
EID=071382660 LN=STEVENS SAL= 10000.00
EID=112847612 LN=SMITH SAL= 13200.00
EID=117593129 LN=JONES SAL= 18480.00
.
.
.
EID=818692173 LN=CROSS SAL= 25775.00
EOF:
```

The Show List, or subtree, consists of all segment instances that have data for all of the fields specified (Employee Identification Number, Last Name and Salary). Records lacking instances of any of these fields (for example, short-path records) are not included in the list.

Example **Selecting All Fields Between Two Named Fields**

```
show emp_id * bank_name
type 2
  EID=071382660 LN=STEVENS  FN=ALFRED  HDT=800602
  DPT=PRODUCTION  CSAL=11000.00  CJC=A07  OJT= 25.00  BN=
  EID=112847612 LN=SMITH    FN=MARY    HDT=810701
  DPT=MIS          CSAL=13200.00  CJC=B14  OJT= 36.00  BN=
```

All fields between (and including) EMP_ID and BANK_NAME are included in the Show List. (Stevens and Smith do not have a bank for electronic transfer and, therefore, the value for BN is blank.)

Example **Selecting All Fields**

To select all fields, use an asterisk instead of field names.

```
SHOW *
```

Note: To examine the contents of the current position in the Show List, you can use TYPE to print just the fields named in the SHOW subcommand. Use DISPLAY or CRTFORM if you wish to see the contents of other fields in the selected segments. (Use TYPE with CRTFORM to see the display.)

Subsequent navigation keywords will show the field values for the current position for each of the fields named in the SHOW subcommand.

Reference **Commands Similar to SHOW**

None.

TLOCATE Command

TLOCATE is a convenience feature that combines the capabilities of the LOCATE subcommand with those of TOP. When issued, the search begins at the top of the database. This combined functionality allows you to automate processes more easily using the X and Y subcommands.

If the subcommand AGAIN is used following TLOCATE, it locates the same record rather than moving ahead to the next instance as it would with LOCATE.

Syntax **TLOCATE Command**

TLocate field rel value [[AND|,]field rel value [,,\$] [|nn]]*

where:

field

Is the field name of the target(s).

rel

Is one of the following test relations:

Relation	Meaning
EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS	Contains
OMITS	Omits

value

Is the object of the comparison.

The comma-dollar sign (,\$) terminator character is not required if only one record is sought. However, it is required if you provide a replication factor larger than one. If the replication factor is set to *, then all records meeting the test conditions are displayed from the current position to the end of the database.

When using more than one test relation, separate them either with commas or the word AND, as follows

locate field rel value, field rel value

or:

locate field rel value AND field rel value

If you supply a list of values with an EQ test, separate the values with the word OR:

locate field EQ value OR value OR value

Example Using the TLOCATE Command

```
show last_name first_name department
tlocate dpt=production
  LN=STEVENS  FN=ALFRED  DPT=PRODUCTION
next 5
  LN=IRVING   FN=JOAN    DPT=PRODUCTION
tlocate dpt=production
  LN=STEVENS  FN=ALFRED  DPT=PRODUCTION
```

Reference Commands Similar to TLOCATE

LOCATE is the same command, but without the TOP function.

TOP Command

The current position is set at the first logical record in the database. If the next subcommand is TYPE or NEXT, the first record is retrieved and displayed.

When the message EOF: appears after any subcommand, use TOP to reset the current position.

Syntax TOP Command

```
TOp
```

Example Using the TOP Command

```
show emp_id last_name salary
next 30
  EOF:
top
  TOF:
next
  EID=071382660 LN=STEVENS SAL= 11000.00
```

The current position is reset to the top of the database.

Reference Commands Similar to TOP

- SHOW also takes you to the top of the database, but its primary purpose is the selection of the logical database view that you wish to use.
- TLOCATE goes to the top of the database before starting its search for the field(s) you have specified.

TYPE Command

The TYPE command displays the values of the named fields or displays the contents of a CRTFORM.

Syntax **TYPE Command**

TYpe [*factor*]

where:

factor

Is one of the following:

1 is the default.

n displays the record at the current position plus the next *n*-1 records, if the replication factor is greater than 1.

* displays the message EOF: after the last record in the database is displayed. Use TOP to reset the current position to the top of the database.

Example **Using the TYPE Command**

```
show emp_id last_name salary
type 6
EID=071382660 LN=STEVENS SAL= 11000.00
EID=071382660 LN=STEVENS SAL= 10000.00
EID=112847612 LN=SMITH SAL= 13200.00
EID=117593129 LN=JONES SAL= 18480.00
EID=117593129 LN=JONES SAL= 17750.00
EID=119265415 LN=SMITH SAL= 9500.00
```

The record at the current position and the next five records are displayed.

Reference **Commands Similar to TYPE**

- The DISPLAY command also shows the contents of the currently active data fields, but DISPLAY shows all the named fields in a neat vertical list, whether they are in the SHOW command or not.
- CRTFORM is used to format a screen, showing the full field names and the field comments, blocked two to a line. Use TYPE to show the contents of the CRTFORM.

UP Command

The UP subcommand resets the current position to the first descendant instance under a parent instance. Hence, it moves the position to the start of the current chain.

Syntax UP Command

```
UP fieldname
```

where:

```
fieldname
```

Is the name of a field in a descendant segment.

Example Using the UP Command

```
show emp_id last_name salary pay_date
next 5
  EID=071382660 LN=STEVENS  SAL= 10000.00 PD=820630
up pay_date
  EID=071382660 LN=STEVENS  SAL= 10000.00 PD=820528
```

The current position is reset to the first instance of PAY_DATE information for Stevens.

Reference Commands Similar to UP

None.

X and Y Commands

The X and Y subcommands are used to store a complete SCAN subcommand for later execution by simply typing in the appropriate letter (X or Y).

To set, but not execute, a value for X or Y, type it as a first letter in front of any other subcommand. Any print suppression control, and the replication factors, are picked up from the stored subcommand.

Syntax X and Y Commands

```
[x|y] subcommand
```

Example Using the X and Y Commands

```
y display emp_id last_name curr_sal pay_date gross
show emp_id pay_date
next
  EID=071382660 PD=820831
y
EMP_ID      =071382660
LAST_NAME   =STEVENS
CURR_SAL    = 11000.00
PAY_DATE    =820831
GROSS       =  916.67
```

A series of operations can be performed by repeatedly entering X and Y subcommands.

Reference Commands Similar to X and Y

None.

? Command

The ? subcommand recalls and displays the last recognized subcommand issued in the SCAN mode.

Syntax ? Command

?

Example Using the ? Command

```
Show emp_id last_name salary jobcode
locate dpt=mis
  EID=112847612 LN=SMITH SAL= 13200.00 JBC=B14
?
  LOCATE DPT=MIS
again
  EID=117593129 LN=JONES SAL= 18480.00 JBC=B03
```

Here the LOCATE operation returns a record. AGAIN locates the next record that meets the stated criteria.

Reference Commands Similar to ?

None.

Subcommand Summary

CHAPTER 13

Directly Editing FOCUS Databases With FSCAN

Topics:

- Introduction
- Entering FSCAN
- Using FSCAN
- The FSCAN Facility and FOCUS Structures
- Scrolling the Screen
- Selecting a Specific Instance by Defining a Current Instance
- Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands
- Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands
- Modifying the Database
- Repeating a Command: ? and =
- Saving Changes: The SAVE Without Exiting FSCAN Command
- Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands
- The FSCAN HELP Facility
- Syntax Summary

The full-screen FSCAN facility enables you to edit FOCUS databases directly on your terminal screen. You can use FSCAN to add, update, and delete data from FOCUS databases as if the segments in the FOCUS databases were flat files on a full-screen editor. You can type over field values, or change them by issuing commands.

Introduction

FSCAN enables you to:

- Add records to new or existing FOCUS databases.
- Change field values in FOCUS databases. With FSCAN you can change the values in key fields (not possible with MODIFY requests).
- Delete records from FOCUS databases.
- Search through FOCUS databases to locate instances of specified character strings or values.

If your database is protected by shadow paging, the changes you make on FSCAN are not permanent until you issue a command to do so. You may choose to exit FSCAN without saving any of the changes.

Databases on Which FSCAN Can Operate

FSCAN can operate on databases having the following attributes:

- The databases are FOCUS databases, not databases of other types.
- The databases are individual databases, not combined structures created by the COMBINE command.
- The length of the root key field in the database does not exceed 61 bytes, and the sum of the field name length plus the field length does not exceed 73 bytes.

Also, note the following regarding databases:

- FSCAN does not accept alternate file views.
- Databases that you specify with the USE command using the READ option are write protected.
- Databases that you are viewing on a FOCUS Database Server in Simultaneous Usage mode are write protected.

Segments on Which FSCAN Can Operate

The following rules apply to the display and editing of segments in FSCAN:

- FSCAN does not display a segment containing a key field longer than 61 bytes and the sum of the field name length plus the field length does not exceed 73 bytes, nor does it display the descendants of that segment.
- When you input a new segment instance, the instance must have a key unique to its group. (In the root segment, this means all the instances in the segment; in a descendant segment, this means all the instances that share a parent instance). If you try to input an instance with a duplicate key, FSCAN will generate an error message.
- If you change a key field value of an instance, the new instance key (the combination of all key field values in the instance) must be unique to the group. If you try to change the key to a duplicate, FSCAN will generate an error message.
- If you use FSCAN on segments already containing duplicate keys, the results are unpredictable. If the root segment has duplicate keys, an attempt to display a screen with these duplicates results in FSCAN terminating in an error. If a descendant segment has duplicate keys, an FSCAN error is displayed and you are positioned at the parent segment.
- When a segment is type S0 or blank, no one field is designated as the key field. FSCAN considers all fields in such segments to be key fields. This has two ramifications:
 - You cannot input a segment instance that is the duplicate of another in the same group.
 - You cannot update a segment instance so that it duplicates another segment instance in the same group.

Fields That FSCAN Can Display

FSCAN can display fields containing the following attributes:

- The field length does not exceed 61 bytes and the sum of the field name length plus the field length does not exceed 73 bytes.
- The fields are real database fields, not DEFINEd fields.
- FSCAN displays group fields as their individual members, not as a group.

Note: Text fields cannot be displayed in FSCAN.

Database Integrity Considerations

How FSCAN treats the changes you make to the database depends on whether the database is protected by shadow paging.

If you are using shadow paging, FSCAN writes your changes to a shadow database. If you enter the commands END, FILE, or SAVE, the changes become part of the real database. If you enter the command QUIT or if FSCAN terminates abnormally, the changes disappear and the database is not affected.

If you are not using shadow paging, FSCAN writes your changes directly to the database. The changes remain even after you enter the QUIT command.

In CMS, the operating system performs shadow paging automatically. In MVS, FOCUS performs shadow paging using the Absolute File Integrity facility.

DBA Considerations

If the database is protected by the DBA security facility, then the ACCESS attribute in the Master File restricts users in the following way:

- Users with read-write access (ACCESS=RW) and write-only access (ACCESS=W) have unrestricted access to the database, with the exception of what is denied them by the RESTRICT and NAME attributes.
- Users with update-only access (ACCESS=U) can display the entire database, with the exception of what is denied them by the RESTRICT and NAME attributes. However, they cannot input or delete instances and can only update non-key fields.
- Users with read-only access (ACCESS=R) to any part of the database cannot use FSCAN on the database.

FSCAN honors DBA security restrictions on segments and fields. FSCAN does not display those segments and fields from which the user is restricted. FSCAN does not honor DBA field value restrictions and will display all field values regardless of the user.

If the user has no access to a key field in the root segment, that user is blocked from using FSCAN on the database.

If the user has no access to a segment, that segment is not listed on the menu that appears when the user enters the CHILD command.

Entering FSCAN

Enter the full-screen FSCAN facility from FOCUS with

```
FSCAN FILE filename
```

where:

```
filename
```

Is the name of the database you are editing. The database must be a FOCUS database. You may also enter FSCAN by typing:

```
FS FILE filename
```

For example, to edit the EMPLOYEE database, enter:

```
FSCAN FILE EMPLOYEE
```

Entering FSCAN With a SHOW List

By default, FSCAN makes all fields in the database available to the user. However, it is possible to restrict the fields available with the SHOW option.

Syntax How to Enter FSCAN With a SHOW List

```
FSCAN FILE filename SHOW  
[fieldname.....fieldname.... | SEG.fieldname]  
END
```

where:

```
SHOW
```

Indicates that specific fields will be displayed. The SHOW keyword must appear on the same line as the FSCAN command.

```
fieldname...
```

Are the fields to be displayed.

```
END
```

Is required and must be specified on a line by itself.

Example Entering FSCAN With a SHOW List

For example, the commands

```
FSCAN FILE EMPLOYEE SHOW  
EMP_ID LAST_NAME FIRST_NAME SEG.GROSS  
END
```

would provide access to only the selected fields in the root segment and to the whole segment containing the field GROSS. The above commands would produce the following display:

FSCAN	FILE	EMPLOYEE	FOCUS	A	CHANGES	:0
		EMP_ID		LAST_NAME		FIRST_NAME
====	-----	-----	-----	-----		-----
==		071382660		STEVENS		ALFRED
==		112847612		SMITH		MARY
==		117593129		JONES		DIANE
==		119265415		SMITH		RICHARD
==		119329144		BANNING		JOHN
==		123764317		IRVING		JOAN
==		126724188		ROMANS		ANTHONY
==		219984371		MCCOY		JOHN
==		326179357		BLACKWOOD		ROSEMARIE
==		451123478		MCKNIGHT		ROGER
==		543729165		GREENSPAN		MARY
-----INPUT-----						
==						
==>						

MORE=>

The only child segment that can be displayed is the SALINFO segment, which contains the field GROSS.

Allowing Uppercase and Lowercase Alpha Fields

By default, FSCAN translates all input and changed alpha fields to uppercase. If uppercase and lowercase input and updates are to be respected, then enter FSCAN with the LOWER keyword.

Syntax How to Specify Case Sensitivity in FSCAN

```
FSCAN FILE filename [case]
```

where:

case

Is one of the following:

UPPER translates all input and changed alpha fields into uppercase. UPPER is the default.

LOWER preserves uppercase and lowercase input and is analogous to the CRTFORM LOWER statement in MODIFY.

MIXED is a synonym for LOWER.

Using FSCAN

When you enter FSCAN, FSCAN displays as much as it can of the root segment of the data source. For example, if you view the EMPLOYEE data source with FSCAN, using the following command

```
FSCAN FILE EMPLOYEE
```

you will see the following screen:

```
1. FSCAN FILE      EMPLOYEEFOCUS      A1              CHANGES :      0

2. EMP_ID          LAST_NAME          FIRST_NAME        HIRE_DATE        DEPARTMENT
   -----          -
3. == 071382660    STEVENS            ALFRED            800602           PRODUCTION
4. == 112847612    SMITH              MARY              810701           MIS
   == 117593129    JONES              DIANE             820501           MIS
   == 119265415    SMITH              RICHARD           820104           PRODUCTION
   == 119329144    BANNING            JOHN              820801           PRODUCTION
   == 123764317    IRVING             JOAN              820104           PRODUCTION
   == 126724188    ROMANS             ANTHONY           820701           PRODUCTION
   == 219984371    MCCOY              JOHN              810701           MIS
   == 326179357    BLACKWOOD          ROSEMARIE         820401           MIS
   == 451123478    MCKNIGHT           ROGER             820202           PRODUCTION
   -----          -
5. ==              --INPUT-----
6. ==>
7.                                                         MORE=>
```

This screen displays the contents of the root segment of the EMPLOYEE database. Each record on the screen is one instance in the root segment. The numbers in the diagram refer to the notes below:

1. The header shows the name of the database and the number of changes made to the database since the last save.
2. Each field is labeled with a column heading.
3. The first record at the top of the screen is called the current instance. Many commands operate only on this record. When you first enter FSCAN, this record is the first instance in the root segment.
4. The equal signs (=) in the left margin of the screen indicate the prefix area. This is where you enter prefix area commands.
The key field value in each record appears highlighted.
5. The last line with equal signs is called the input area and is reserved exclusively for input.
6. The arrow at the lower-left corner of the screen points to the command line. This is where you enter FSCAN commands.
7. The MORE symbol at the lower-right corner of the screen indicates that each record extends to the right of the screen.

This section discusses various functions of the FSCAN facility. For an alphabetic summary of commands, see *Syntax Summary* on page 13-40.

The FSCAN facility displays one segment at one time. (For the root segment, FSCAN displays all instances in the segment; for descendant segments, FSCAN displays all instances sharing the same parent instance.) Each record on the screen is one segment instance. The first instance at the top of the screen is called the current instance.

The FSCAN facility also displays segments in SINGLE mode, that is, one instance at one time. SINGLE mode is discussed in *Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands* on page 13-26.

Note the different types of commands:

- **Prefix area commands** are typed in the prefix area on the left of the screen display. Prefix area commands operate only on the line where they are typed.
- **Command-line commands** are typed on the command line at the bottom of the screen. Some commands operate on the entire screen, others operate only on the current instance at the top of the screen. There are two types of command-line commands:
 - **Immediate commands.** When you execute an immediate command, the database remains unchanged even if you typed changes on the screen. There are five immediate commands:
`LEFT`
`RIGHT`
`RESET`
`?`
`QQUIT`
 - **Non-immediate commands.** When you execute a non-immediate command, any changes you type on the screen will be written to the database even if the command itself does not modify the database.

The following rules apply to commands:

- You may use unique truncations for commands. When this section specifies a command syntax, the unique truncation is shown in uppercase.
- Commands that use field names as parameters require the full field name, alias, or unique truncation.
- You may enter two commands at one time by separating the commands with a semicolon. For example, to enter the commands NEXT 5 and CHILD at one time, type:

```
NEXT 5; CHILD
```

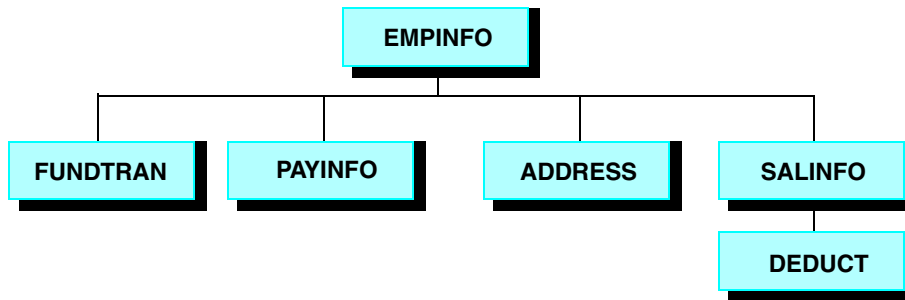
The FSCAN Facility and FOCUS Structures

This section is a brief summary of FOCUS structures and how they affect the FSCAN facility.

FOCUS databases are organized into segments which have the following properties:

- Segments consist of individual data records called segment instances, in which fields have a one-to-one correspondence with each other.
- Segments relate to each other as parents and children.
- A group of instances in a child segment describes one instance in a parent segment.
- One parent segment may have many child segments, but a child segment may have only one parent.
- A FOCUS structure has one segment from which all other segments are descended. This is called the root segment.

The diagram below represents the structure of the EMPLOYEE database:



Note the position of the segments in the structure:

- The EMPINFO segment is the root segment. All other segments are descended from it.
- EMPINFO has four children: the FUNDTRAN, PAYINFO, ADDRESS, and SALINFO segments.
- The SALINFO segment has one child, the DEDUCT segment.

The FSCAN facility displays instances in one segment at one time. When it displays the root segment (as it will when you first enter FSCAN), it displays all the instances in the segment. The following screen illustrates how FSCAN displays the EMPINFO segment.

FSCAN	FILE	EMPLOYEEFOCUS	A1	CHANGES : 0	
	EMP_ID	LAST_NAME	FIRST_NAME	HIRE_DATE	DEPARTMENT
==	071382660	STEVENS	ALFRED	800602	PRODUCTION
==	112847612	SMITH	MARY	810701	MIS
==	117593129	JONES	DIANE	820501	MIS
==	119265415	SMITH	RICHARD	820104	PRODUCTION
==	119329144	BANNING	JOHN	820801	PRODUCTION
==	123764317	IRVING	JOAN	820104	PRODUCTION
==	126724188	ROMANS	ANTHONY	820701	PRODUCTION
==	219984371	MCCOY	JOHN	810701	MIS
==	326179357	BLACKWOOD	ROSEMARIE	820401	MIS
==	451123478	MCKNIGHT	ROGER	820202	PRODUCTION
-----INPUT-----					
--					
==					
==>					

MORE=>

Note that the screen only displays the first five fields of the first ten instances in the segment. To view the other fields and instances, use the scrolling facilities described in *Scrolling the Screen* on page 13-14.

Also note that you cannot move from one segment to another by simply scrolling. To move from a parent segment to a child segment and back again, you must use the PARENT and CHILD commands discussed in *Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 13-23.

When FSCAN displays a child segment, it displays only those instances relating to an instance in the parent segment. You can scroll back and forth to view all the instances in the group, but you cannot scroll to view the child instances of another parent. At the top of the screen, FSCAN displays up to five keys of the parent instance, and of the parent of the parent, and so on, up to the root segment.

For example, the EMPINFO segment contains the ID numbers and names of employees; its child (SALINFO) contains monthly pay instances. (Each instance lists how much each employee was paid each month.) Each group of instances in SALINFO represents all the monthly pay of one employee recorded in the EMPINFO segment. When FSCAN displays the SALINFO segment, it displays one group of instances at one time.

This is how FSCAN displays the monthly pay of Alfred Stevens, who is listed in the EMPINFO segment. Note that Mr. Stevens' employee ID (the EMPINFO key field) appears at the top of the screen:

```
FSCAN  FILE  EMPLOYEEFOCUS  A1  CHANGES : 0

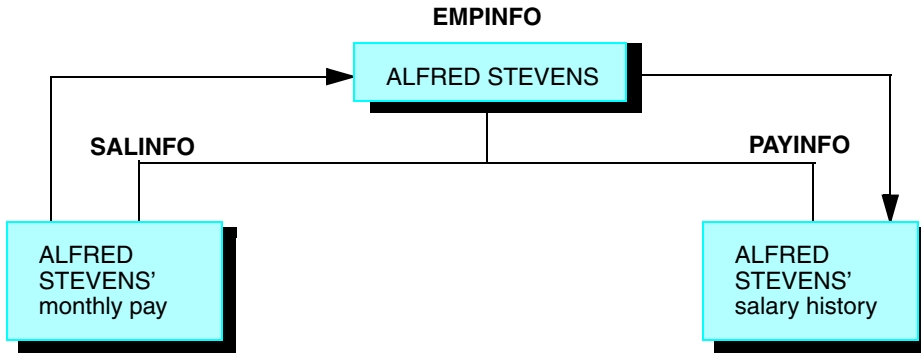
EMP_ID : 071382660

      PAY_DATE      GROSS
      -----      -
==      820831      916.67
==      820730      916.67
==      820630      916.67
==      820528      916.67
==      820430      916.67
==      820331      916.67
==      820226      916.67
==      820129      916.67
==      811231      833.33
-----INPUT-----
--
==

==>
```

If you are displaying one child segment and wish to display another one, you must return to the parent and request the other child segment. For example, if you are examining Alfred Stevens' monthly pay and wish to view his salary history (contained in the segment PAYINFO), return to the EMPINFO segment and request PAYINFO information for Alfred Stevens using the CHILD command described in *Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 13-23.

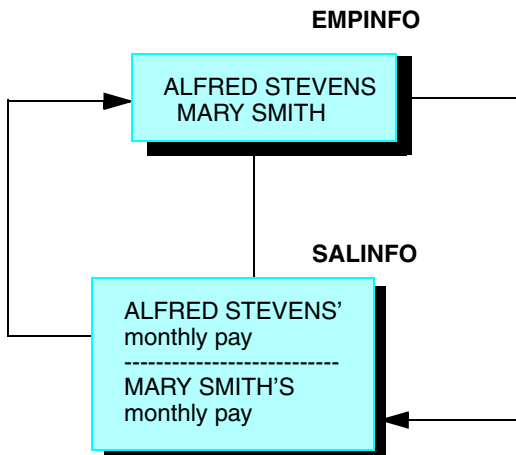
The figure below shows this path schematically. The arrows show the direction you are traveling to move from the SALINFO segment to the PAYINFO segment:



Similarly, if you are displaying one group of child instances and wish to display another group within the same segment but belonging to a different instance in the parent, you must return to the parent segment and request the child segment for the other instance.

For example, suppose you are examining Alfred Stevens' monthly pay and wish to view Mary Smith's monthly pay. You must return to the **EMPINFO** segment and select the **SALINFO** segment for Mary Smith.

The figure below shows this path schematically. The arrows show the direction you are traveling to move from Alfred Stevens' monthly pay instances to Mary Smith's monthly pay instances:



Scrolling the Screen

You may scroll the screen forward and backward, right and left.

Syntax How to Scroll the Screen Forward

To scroll forward one screen in a segment, enter

`FOrward`

or press the PF8 or PF20 key. Note that the last instance on one screen becomes the first instance on the next screen.

To scroll the screen *n* lines forward, enter

`Next n`

or:

`DOwn n`

If you do not enter a number for *n*, the default is 1.

Example Scrolling Forward

For example, suppose the screen displays the EMPLOYEE root segment as shown below.

```

FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :0

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
== 071382660    STEVENS    ALFRED      800602     PRODUCTION
== 112847612    SMITH      MARY        810701     MIS
== 117593129    JONES      DIANE       820501     MIS
== 119265415    SMITH      RICHARD     820104     PRODUCTION
== 119329144    BANNING    JOHN        820801     PRODUCTION
== 123764317    IRVING     JOAN        820104     PRODUCTION
== 126724188    ROMANS     ANTHONY     820701     PRODUCTION
== 219984371    MCCOY      JOHN        810701     MIS
== 326179357    BLACKWOOD  ROSEMARIE   820401     MIS
== 451123478    MCKNIGHT   ROGER       820202     PRODUCTION
-----INPUT-----
==
==> forward

                                                                MORE=>
    
```


When you type the FORWARD command on the command line and press Enter, the following screen appears:

```

FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :0

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
==  451123478    MCKNIGHT   ROGER       820202     PRODUCTION
==  543729165    GREENSPAN  MARY        820401     MIS
==  818692173    CROSS     BARBARA     811102     MIS

-----INPUT-----
==
==>
MORE=>

```

Syntax How to Scroll the Screen Backward

To scroll the screen backward, enter

`Backward`

or press the PF7 or PF19 key.

Syntax How to Scroll the Screen to the Right and the Left

To scroll the screen one panel to the right, enter

`RIght`

or press the PF11 or PF23 key.

To scroll the screen one panel to the left, enter

`LEft`

or press the PF10 or PF22 key.

The commands RIGHT and LEFT are immediate commands. When you scroll right and left, FSCAN does not enter changes you typed on the screen until you press Enter after scrolling.

Example Scrolling the Screen

For example, if you scroll the EMPLOYEE root segment display one panel to the right, the following screen appears:

FSCAN FILE EMPLOYEEFOCUS A1			CHANGES :0
CURR_SAL	CURR_JOBCODE	ED_HRS	
== 11000.00	A07	25.00	
== 13200.00	B14	36.00	
== 18480.00	B03	50.00	
== 9500.00	A01	10.00	
== 29700.00	A17	.00	
== 26862.00	A15	30.00	
== 21120.00	B04	5.00	
== 18480.00	B02	.00	
== 21780.00	B04	75.00	
== 16100.00	B02	50.00	
-----INPUT-----			
==			
==>			

MORE=>

Selecting a Specific Instance by Defining a Current Instance

This section describes how to move through the database by defining a particular instance as the current instance. The current instance is always the top instance on the screen. Certain commands only operate on the current instance.

Procedure How to Define a Current Instance

To define an instance as the current instance, type a slash (/) in the prefix area corresponding to the instance.

You may also type a slash before or after the following prefix area commands:

- The K command (K/ or /K). After FSCAN changes the key field and displays the instance in proper sequence, it makes the instance the current instance.
- The I command (I/ or /I). After FSCAN adds a new instance to the database, it makes the instance the current instance.

Example Defining a Current Instance: The "/" Prefix

For example, suppose you type a slash in the prefix area of John Banning's instance, as shown below:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0

      EMP_ID      LAST_NAME      FIRST_NAME      HIRE_DATE      DEPARTMENT
      -----      -
== 071382660    STEVENS        ALFRED          800602         PRODUCTION
== 112847612    SMITH          MARY            810701         MIS
== 117593129    JONES         DIANE           820501         MIS
== 119265415    SMITH         RICHARD         820104         PRODUCTION
/= 119329144    BANNING       JOHN            820801         PRODUCTION
== 123764317    IRVING        JOAN            820104         PRODUCTION
== 126724188    ROMANS        ANTHONY         820701         PRODUCTION
== 219984371    MCCOY         JOHN            810701         MIS
== 326179357    BLACKWOOD     ROSEMARIE      820401         MIS
== 451123478    MCKNIGHT     ROGER           820202         PRODUCTION
-----INPUT-----
==
==>
MORE=>

```

When you press Enter, the following screen appears:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0

      EMP_ID      LAST_NAME      FIRST_NAME      HIRE_DATE      DEPARTMENT
      -----      -
== 119329144    BANNING       JOHN            820801         PRODUCTION
== 123764317    IRVING        JOAN            820104         PRODUCTION
== 126724188    ROMANS        ANTHONY         820701         PRODUCTION
== 219984371    MCCOY         JOHN            810701         MIS
== 326179357    BLACKWOOD     ROSEMARIE      820401         MIS
== 451123478    MCKNIGHT     ROGER           820202         PRODUCTION
== 543729165    GREENSPAN     MARY            820401         MIS
== 818692173    CROSS         BARBARA         811102         MIS
-----INPUT-----
==
==>
MORE=

```

Syntax **How to Define the First and Last Instances of a Segment on Display: The FIRST, LAST, and TOP Commands**

FSCAN displays all instances in a segment that share a common parent instance. For the root segment, this means all the instances in the segment. To define the first instance in the group as the current instance, enter:

`FIRST`

If you are displaying instances in the root segment, FIRST will make the first instance in the database the current instance. If you are displaying instances in a child segment and use the FIRST command, the first child instance will become the current instance.

To define the last instance as the current instance, enter:

`LAST`

To select the first instance in the root segment of the database to be the current instance, enter:

`TOP`

TOP displays the root segment, scrolled to the leftmost panel, with the first instance the current instance.

Example **Defining the Last Instance as the Current Instance With LAST**

For example, if you enter LAST on the EMPLOYEE root segment display, the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0

  EMP_ID    LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
  -----
== 818692173 CROSS      BARBARA     811102     MIS

-----INPUT-----
==
==>

MORE=>
```

Syntax**How to Locate an Instance Based on Field Values: The LOCATE Command**

LOCATE searches for instances containing field values that fulfill certain conditions. For example, it can search for an instance with a LAST_NAME value of BANNING or a CURR_SAL value less than 20,000. LOCATE searches starting with the current instance.

The syntax is (entered on one line)

```
Locate field1 rel1 value1 [OR value1a OR value1b OR ...]
```

```
 [{AND|,} field2 rel2 value2 {AND|,} ...]
```

where:

fieldn ...

Is a field to be tested.

reln ...

Is one of the following condition relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance with a field value that passes the test becomes the current segment.

If you supply more than one test condition in the command, FSCAN searches for the instance that fulfills all of the conditions. Separate the test conditions in the command with the word AND or with a comma (,).

OR

Enables you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in a single LOCATE command.

Selecting a Specific Instance by Defining a Current Instance

The LOCATE command searches starting with the first instance following the current instance. If LOCATE cannot find the instance, it displays a message and the current instance does not change.

Example Locating an Instance Based on Field Values

For example, suppose the first instance in the EMPLOYEE root segment is the current instance. If you issue the command

```
LOCATE LAST_NAME EQ SMITH
```

the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1          CHANGES : 0
```

EMP_ID	LAST_NAME	FIRST_NAME	HIRE_DATE	DEPARTMENT
== 112847612	SMITH	MARY	810701	MIS
== 117593129	JONES	DIANE	820501	MIS
== 119265415	SMITH	RICHARD	820104	PRODUCTION
== 119329144	BANNING	JOHN	820801	PRODUCTION
== 123764317	IRVING	JOAN	820104	PRODUCTION
== 126724188	ROMANS	ANTHONY	820701	PRODUCTION
== 219984371	MCCOY	JOHN	810701	MIS
== 326179357	BLACKWOOD	ROSEMARIE	820401	MIS
== 451123478	MCKNIGHT	ROGER	820202	PRODUCTION
== 543729165	GREENSPAN	MARY	820401	MIS

```
-----INPUT-----  
==  
==> MORE=>
```

These are other examples of the LOCATE command:

```
LOCATE JOBCODE EQ A07 OR A17
```

This LOCATE searches for the first segment instance that has a JOBCODE value of either A07 or A17.

```
LOCATE LAST_NAME CO WOOD
```

This LOCATE searches for the first segment instance with a LAST_NAME value that contains the character string WOOD.

```
LOCATE HIRE_DATE GT 820401 AND JOBCODE IS B02 OR B03
```

This LOCATE searches for the first segment instance with both a HIRE_DATE value greater than 820401 and a JOBCODE value that is either B02 or B03.

Syntax**How to Find an Instance in a Group: The FIND Command**

The FIND command works within the group of instances being displayed. In the root segment, this is all instances in the segment; in descendant segments, this is all instances sharing a common parent instance. FIND searches for instances containing field values that fulfill certain conditions. For example, it can search for an instance with a LAST_NAME value of BANNING or a CURR_SAL value less than 20,000. FIND searches starting with the current instance.

The syntax is entered on one line.

```
FIND field1 rel1 value1 [OR value1a OR value1b OR ...]
[{AND|,} field2 rel2 value2 {AND|,} ...]
```

where:

fieldn ...

Is a field in the segment.

reln ...

Is one of the following condition relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance with a field value that passes the test becomes the current segment.

If you supply more than one test condition in the command, FSCAN searches for the instance that fulfills all of the conditions. Separate the test conditions in the command with the word AND or with a comma (,).

OR

Enables you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in a single FIND command.

Selecting a Specific Instance by Defining a Current Instance

The FIND command searches the group starting with the first instance following the current instance. To search the entire group, issue the FIRST command before issuing FIND. If FIND cannot find the instance, it displays a message and the current instance does not change.

Example Finding an Instance in a Group

For example, suppose the first instance in the EMPLOYEE root segment is the current instance. If you issue the command

```
FIND LAST_NAME EQ SMITH
```

the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0
```

EMP_ID	LAST_NAME	FIRST_NAME	HIRE_DATE	DEPARTMENT
== 112847612	SMITH	MARY	810701	MIS
== 117593129	JONES	DIANE	820501	MIS
== 119265415	SMITH	RICHARD	820104	PRODUCTION
== 119329144	BANNING	JOHN	820801	PRODUCTION
== 123764317	IRVING	JOAN	820104	PRODUCTION
== 126724188	ROMANS	ANTHONY	820701	PRODUCTION
== 219984371	MCCOY	JOHN	810701	MIS
== 326179357	BLACKWOOD	ROSEMARIE	820401	MIS
== 451123478	MCKNIGHT	ROGER	820202	PRODUCTION
== 543729165	GREENSPAN	MARY	820401	MIS

```
-----INPUT-----  
==  
==>  
  
MORE=>
```

These are other examples of the FIND command:

```
FIND DEPARTMENT EQ MIS OR SALES
```

This FIND searches for the first segment instance that has a DEPARTMENT value of either MIS or SALES.

```
FIND LAST_NAME CO WOOD
```

This FIND searches for the first segment instance with a LAST_NAME value that contains the character string WOOD.

```
FIND HIRE_DATE GT 820401 AND DEPARTMENT EQ MIS OR PRODUCTION
```

This FIND searches for the first segment instance with both a HIRE_DATE value greater than 820401 and a DEPARTMENT value that is either MIS or PRODUCTION.

Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands

The CHILD, PARENT, and JUMP commands enable you to display the data in different segments of a data source.

Syntax **How to Display a Child Segment**

To display instances in a child segment relating to the current instance, enter

`CHiLd`

or press PF5 or PF17. If the segment on the screen when you enter the command has only one child segment, FSCAN shows the child segment. If the segment on the screen has more than one child segment, FSCAN displays a menu of child segments. Select a segment by entering its number. (**Note:** The menu does not display segments restricted to you as a result of DBA restrictions.)

If you already know the number of the segment on the menu, you can skip the menu by entering

`CHiLd n`

where:

`n`

is the number of the segment on the menu.

You can display the child instances of any instance on the screen by typing C in the prefix area next to the instance. You can skip the menu by typing C followed by the number of the segment on the menu.

Example **Displaying a Child Segment**

For example, suppose you are displaying the root segment of the EMPLOYEE database and you want to see the monthly pay of Mary Smith. Monthly pay is contained in the segment SALINFO, a child of the root segment. First, make Mary Smith's instance the current instance. Then, enter the command:

CHILD

The following menu appears:

```
FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0

Please enter the number of the child segment you want

1) FUNDTRAN                2) PAYINFO
3) ADDRESS                 4) SALINFO

=> Enter the number of the child you want
    Enter 0 to stay at parent.
```

Enter the number 4. The following screen appears:

```

FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :0

EMP_ID : 112847612

          PAY_DATE          GROSS
          -----          -
==      820831          1100.00
==      820730          1100.00
==      820630          1100.00
==      820528          1100.00
==      820430          1100.00
==      820331          1100.00
==      820226          1100.00
==      820129          1100.00

-----INPUT-----
==
==>
    
```

Note that the header displays the key field value of the parent instance. Since EMP_ID is the key field of the root segment, the header displays Mary Smith’s employee ID.

Also, you could have gone directly from the EMPLOYEE root segment to the monthly pay segment by doing one of the following:

- Typing CHILD 4 on the command line.
- Typing C4 in the prefix area.

Syntax **How to Display the Parent Segment**

To return to the parent segment, enter

Parent

or press PF4 or PF16. The current instance in the parent is the same as before you entered the CHILD command or C prefix area command.

Syntax **How to Display the First Child of the Next Parent Instance**

To move to the first child of the next parent instance, enter

JUMP

or press PF12 or PF24 while FSCAN is displaying a child segment.

Example **Displaying the First Child of the Next Parent Instance**

For example, if you enter JUMP while the PAYINFO segment is being displayed for a particular employee, the PAYINFO segment for the next employee in the EMP_INFO segment is displayed. JUMP may be issued anywhere.

Displaying a Single Instance on One Screen: The *SINGLE* and *MULTIPLE* Commands

To display a single instance on the screen, enter:

*S*ingle

This places you in SINGLE mode. SINGLE mode enables you to view a single segment instance on one screen. Only the current instance appears, but all its fields appear on one screen (unless it has many fields). You may enter all FSCAN commands on the command line at the bottom of the screen, but there is no prefix area. The key field values appear highlighted.

All FSCAN commands (but not prefix area commands) operate in SINGLE mode, except that only one instance is displayed. In particular, note the following:

- If you enter the FORWARD command in SINGLE mode, FSCAN displays the next instance in the segment. If you enter the BACKWARD command, FSCAN displays the previous instance.
- If you enter the CHILD command, only one child instance appears at one time. If you enter the PARENT command, only the parent instance of the current instance appears on the screen.

You can update and delete an instance in SINGLE mode, but you cannot add another instance.

You remain in SINGLE mode until you enter the command:

*M*ultiple

MULTIPLE returns you to normal mode, which displays multiple instances at one time.

Example Using SINGLE Mode

For example, this is how Diane Jones' instance looks in SINGLE mode. Note that there is no input area, and that the arrow at the bottom of the screen points to the command line where you can enter commands:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES : 0

      EMP_ID : 117593129          LAST_NAME : JONES
      FIRST_NAME : DIANE          HIRE_DATE : 820501
      DEPARTMENT : MIS            CURR_SAL : 18480.00
      CURR_JOBCODE : B03          ED_HRS : 50.00

==>

```

Modifying the Database

You may use FSCAN to modify the database by adding, updating, and deleting segment instances.

Adding New Segment Instances: The "I" Prefix

To add a new segment instance to the segment displayed on the screen, type the instance field values in the input area on the bottom of the screen. You can use the Tab key to jump from field to field. Then type I in the prefix area next to the new instance. When you press Enter, FSCAN adds the instance to the database, displaying it in proper sequence based on its key field values.

If the instance you are typing extends beyond the right margin of the screen, use the scrolling commands discussed in *Scrolling the Screen* on page 13-14. FSCAN adds the segment instance when you press Enter or enter any command except RIGHT, LEFT, RESET, ?, and QQUIT.

Note:

- FSCAN does not accept new instances with key field values that are the same as another instance.
- FSCAN does not accept new instances with field values that do not conform to the ACCEPT attribute in the Master File (ACCEPT is explained in the *Describing Data* manual).
- If you want the new instance to become the current instance, type I/ in the prefix area next to the new instance before pressing Enter.

Example Adding New Segment Instances

For example, suppose you want to add Fred Johnson to the EMPLOYEE database, and you want the new instance to become the current instance. Type his instance in the input area as shown below (note the I/ in the prefix area):

```
FSCAN FILE EMPLOYEEFOCUS A1      CHANGES :0

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
==  117593129  JONES      DIANE       820501     MIS
==  119265415  SMITH      RICHARD     820104     PRODUCTION
==  119329144  BANNING    JOHN        820801     PRODUCTION
==  123764317  IRVING     JOAN        820104     PRODUCTION
==  126724188  ROMANS     ANTHONY     820701     PRODUCTION
==  219984371  MCCOY      JOHN        810701     MIS
==  326179357  BLACKWOOD  ROSEMARIE   820401     MIS
==  451123478  MCKNIGHT  ROGER       820202     PRODUCTION
==  543729165  GREENSPAN  MARY        820401     MIS
==  818692173  CROSS     BARBARA     811102     MIS
-----INPUT-----
I/  123123123  johnson    fred        870507     mis

==>

                                         MORE=>
```

When you press Enter, the screen appears as follows:

```

FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :1

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
== 123123123  JOHNSON    FRED        870507     MIS
== 123764317  IRVING     JOAN        820104     PRODUCTION
== 126724188  ROMANS     ANTHONY     820701     PRODUCTION
== 219984371  MCCOY      JOHN        810701     MIS
== 326179357  BLACKWOOD  ROSEMARIE   820401     MIS
== 451123478  MCKNIGHT   ROGER       820202     PRODUCTION
== 543729165  GREENSPAN  MARY        820401     MIS
== 818692173  CROSS      BARBARA     811102     MIS
-----INPUT-----
==

==>
  0 Keys Changed 0 Non-Keys Changed
  0 Records Deleted 1 Records Input

                                     MORE=>

```

If you do not type "I" in the prefix area when you input a new instance, FSCAN displays an error message. To continue, you must do one of the following:

- Enter "I" in the prefix area of the input area.
- Cancel the input by entering the RESET command, typing R in the prefix area, or pressing the PF2 or PF14 key. This also recovers typed-over field values (see the following section).

Note that the RESET command entered on the command line is an immediate command. However, the R prefix-area command is not an immediate command. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Updating Non-Key Field Values

There are three ways to update non-key field values:

- Type over field values.
- Issue the REPLACE command.
- Issue the CHANGE command.

Note that FSCAN does not accept any new field value that does not conform to the ACCEPT attribute in the Master File (the ACCEPT attribute is explained in the *Describing Data* manual).

Procedure How to Type Over Field Values

You may update segment instances by typing over their values on the screen. Use the Tab key to jump from field to field within the same instance.

Example Typing Over Field Values

For example, suppose you want to change Richard Smith’s department from Production to Sales. Simply type over the DEPARTMENT value and press Enter. The screen appears as shown on the next page. Note that the message at the bottom of the screen indicates one changed non-key field.

The screen is:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :0

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
== 071382660 STEVENS   ALFRED     800602     PRODUCTION
== 112847612 SMITH     MARY       810701     MIS
== 117593129 JONES    DIANE      820501     MIS
== 119265415 SMITH    RICHARD    820104     SALES
== 119329144 BANNING  JOHN       820801     PRODUCTION
== 123764317 IRVING   JOAN       820104     PRODUCTION
== 126724188 ROMANS   ANTHONY    820701     PRODUCTION
== 219984371 MCCOY    JOHN       810701     MIS
== 326179357 BLACKWOOD ROSEMARIE  820401     MIS
== 451123478 MCKNIGHT ROGER      820202     PRODUCTION
-----INPUT-----
==
==>
  0 Keys Changed 1 Non-Keys Changed
  0 Records Deleted 0 Records Input

                                     MORE=>
    
```

The message at the bottom of the screen indicates the number of field values you changed since the last time you pressed Enter. The counter at the top of the screen counts the total number of values you changed since the last time the changes were saved on disk.

If you type over field values and change your mind before you press Enter, you can restore the original field values by entering R (to specify the RESET command) on the prefix area next to the instance whose values you are recovering, or by pressing the PF2 or PF14 key. However, if you press Enter before pressing one of these keys, you will not recover the typed-over values.

Note that the RESET command entered on the command line is an immediate command. However, the R prefix area command is not an immediate command. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Syntax **How to Replace Field Values: The REPLACE Command**

The REPLACE command replaces one field value with another either for a specific instance or for all the instances in a group. (In the root segment, this is all the instances in the segment; in a descendant segment, this is all the instances that share a parent instance.) The syntax is

```
REPLACE field1 = value1[,field2 = value2, ...] [,$ {*|n}]
```

where:

fieldn ...

Is a field in the current instance whose value you want to change.

valuen ...

Is a new value for the field.

,\$ {|n}*

Enables you to change multiple instances starting from the current instance (the current instance included). *n* is the number of instances to be searched for the field value you want to change. If you want all instances in the group starting from the current instance changed, use an asterisk (*).

Example **Using REPLACE**

For example, to change Richard Smith's department from Production to Sales, make Richard Smith's instance the current instance. Then enter:

```
REPLACE DEPARTMENT = SALES
```

To change the DEPARTMENT value to SALES in the next five instances, enter:

```
REPLACE DEPARTMENT = SALES,$ 5
```

To change all DEPARTMENT values in the group to SALES, make the first instance on display the current instance by entering:

```
FIRST
```

Then enter:

```
REPLACE DEPARTMENT = SALES,$ *
```

Syntax **How to Change Character Strings Within Field Values: The CHANGE Command**

The CHANGE command changes character strings within field values either for a specific instance or for all the instances in a group (in the root segment, this is all the instances in the segment; in a descendant segment, this is all the instances that share a parent instance). The fields must be alphanumeric. The syntax is

```
CHAnge field = /oldstring/newstring/ [,$ {*|n}]
```

where:

field

Is the name of the field in the current instance whose value you want to change. The field must be alphanumeric, and it cannot be a key field.

oldstring

Is the substring of the field value that you want to change.

newstring

Is the character string to replace the substring.

,\$ {|n}*

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the substring. If you want all instances in the group searched, starting from the current instance, use an asterisk (*).

Example **Using CHANGE**

For example, to change Joan Irving's department from Production to Products, make Joan Irving's instance the current instance. Then enter:

```
CHANGE DEPARTMENT = /ION/S/
```

To change the Production department to Products in the next five instances starting from the current instance, enter:

```
CHANGE DEPARTMENT = /ION/S/,$ 5
```

To change this substring in all the instances in the group, make the first instance on display the current instance by entering:

```
FIRST
```

Then enter:

```
CHANGE DEPARTMENT = /ION/S/,$ *
```

Changing Key Field Values

FSCAN enables you to change values of key fields, either by typing over the values or by using the REPLACE KEY command.

Note: FSCAN does not allow you to change a key field to a value that will make the key field values of one instance the same as another instance.

FSCAN does not accept any new key field value that does not conform to the ACCEPT attribute in the Master File (the ACCEPT attribute is explained in the *Describing Data* manual).

Procedure How to Type Over Key Field Values: The KEY Command

To change the value of a key field, do the following:

1. Type the new value over the old one.
2. Either type a K in the prefix area next to the instance you are changing, or type the command:

Key

If you want the instance to be the current instance after its key value is changed, type *K/* in the prefix area next to the instance.

3. Press Enter.

After you change the key value, FOCUS moves the instance within the segment so that the key values remain sorted in their proper sequence. The screen shows this immediately.

Note: FOCUS does not physically move instances in the root segment, although the instances appear on the FSCAN screen sorted by their key field values.

If you do not enter the KEY command or type K in the prefix area when you change a key field value, FSCAN displays an error message. Before continuing, you must do one of the following:

- Enter the KEY command, or enter K in the prefix area.
- Retype the original key value.
- Restore the key field value by entering the RESET command, typing R in the prefix area, or pressing the PF2 or PF14 key. Other field values you typed over will also be restored

Note: The RESET command entered on the command line is an immediate command. However, the R prefix area command is not an immediate command. If you type any changes on a line that does not specify the R prefix, FSCAN enters the changes.

Example Using KEY

For example, suppose you want to change Alfred Stevens' employee ID from 071382660 to 444555666, and you want his instance to remain the current instance. Type over the employee ID and type K/ in the prefix area.

The screen appears as shown below:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :2

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
k/    444555666  STEVENS    ALFRED      800602     PRODUCTION
==    112847612  SMITH      MARY        810701     MIS
==    117593129  JONES      DIANE       820501     MIS
==    119265415  SMITH      RICHARD     820104     PRODUCTION
==    119329144  BANNING    JOHN        820801     PRODUCTION
==    123764317  IRVING     JOAN        820104     PRODUCTION
==    126724188  ROMANS     ANTHONY     820701     PRODUCTION
==    219984371  MCCOY      JOHN        810701     MIS
==    326179357  BLACKWOOD  ROSEMARIE   820401     MIS
==    451123478  MCKNIGHT   ROGER       820202     PRODUCTION
-----INPUT-----
==
==>
MORE=>

```

When you press Enter, the screen appears as shown below:

```

FSCAN FILE EMPLOYEEFOCUS A1                CHANGES :3

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----      -
==    444555666  STEVENS    ALFRED      800602     PRODUCTION
==    451123478  MCKNIGHT   ROGER       820202     PRODUCTION
==    543729165  GREENSPAN  MARY        820401     MIS
==    818692173  CROSS      BARBARA     811102     MIS
-----INPUT-----
==
==>
  1 Keys Changed 0 Non-Keys Changed
  0 Records Deleted 0 Records Input
MORE=>

```

The message at the bottom of the screen indicates the number of key field values you changed since the last time you pressed Enter.

Syntax **How to Change Key Field Values Using the REPLACE KEY Command**

You may also use the REPLACE command to change key fields of the current instance. The syntax of the REPLACE command to replace key fields is

```
REPLACE KEY key1 = value1[, key2 = value2, ...]
```

where:

keyn ...

Is the key field you want to change. Remember that an instance may have more than one key field (as determined by the SEGTYPE attribute in the Master File).

valuen ...

Is the new value for the key field.

Example **Using REPLACE KEY**

For example, to change Alfred Stevens' employee ID from 444555666 to 071382660, make his instance the current instance by placing a slash in the prefix area, and enter the following:

```
REPLACE KEY EMP_ID = 071382660
```

Deleting Segment Instances: The DELETE Command

You can easily delete a data instance with the DELETE command.

Syntax **How to Delete Segment Instances**

To delete the current instance, type a D in the prefix area next to the instance or enter:

```
DElete
```

FSCAN displays the complete segment instance alone on the screen and asks if you really want to delete it. Press Enter to delete the instance, or respond:

N

Do not delete the current instance. (Returns to the previous screen.)

Q

Do not delete the current instance. (If you made no other changes to the database, entering Q leaves FSCAN and returns to the FOCUS prompt. Otherwise, it returns to the previous screen.)

Note: When you delete an instance, you delete all its descendant instances as well.

Example Using DELETE

For example, suppose you want to delete information about John Banning from the database. First, make John Banning's instance the current instance. Then, enter the DELETE command. The following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :4

      Delete Confirmation Screen

      EMP_ID : 119329144              LAST_NAME : BANNING
      FIRST_NAME : JOHN                HIRE_DATE : 820801
      DEPARTMENT : PRODUCTION          CURR_SAL : 29700.00
      CURR_JOBCODE : A17                ED_HR : .00

==>
      Press ENTER to delete
      Enter N(o) to abort
      Enter Q(uit) to quit session
```

If you press Enter, the screen appears as follows:

```
FSCAN FILE EMPLOYEEFOCUS A1CHANGES :2

      EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      -----
==  123764317  IRVING     JOAN        820104     PRODUCTION
==  126724188  ROMANS     ANTHONY     820701     PRODUCTION
==  219984371  MCCOY      JOHN        810701     MIS
==  326179357  BLACKWOOD ROSEMARIE   820401     MIS
==  451123478  MCKNIGHT  ROGER       820202     PRODUCTION
==  543729165  GREENSPAN MARY        820401     MIS
==  818692173  CROSS     BARBARA     811102     MIS
-----INPUT-----
==
==>

0 Keys Changed 0 Non-Keys Changed
1 Records Deleted 0 Records Input

                                     MORE=>
```

Repeating a Command: ? and =

Two commands help you enter a command repeatedly:

- The ? command displays the last command you entered.
- The = command executes the last command you entered.

Syntax How to Display Previous Commands: The ? Command

To display the last command you entered, enter

?

or press PF6 or PF18. This displays the previous command on the command line. You may then execute the command by pressing Enter or remove the command from the command line.

As you enter FSCAN commands on the command line, FSCAN stores them in a stack in memory. If you enter the ? command repeatedly, FSCAN scrolls through the stack, displaying the commands in stack from the most recent to the oldest.

The ? command is an immediate command. The database remains unchanged until you press Enter a second time or enter a non-immediate command. Immediate commands were explained previously at the beginning of *Using FSCAN* on page 13-7.

Syntax Executing the Previous Command: The = Command

The = command executes the last command you entered. Enter

=

or press PF9 or PF21.

Saving Changes: The SAVE Without Exiting FSCAN Command

To save the changes to the database that you made on FSCAN, enter

SAve

You remain in FSCAN. The counter at the top of the screen that counts changes in the database is reset to 0.

Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands

To exit FSCAN and save the changes you made to the database, enter

`End`

or:

`FIle`

If bad data is encountered upon trying to save your changes, an error message is generated.

To exit FSCAN without saving the changes you made to the database, enter:

`QQuit`

Note: In MVS, QQUIT only suppresses changes made on FSCAN when you are using the Absolute File Integrity facility. Otherwise, FSCAN writes all changes to the database.

If you did not make any changes to the database, you can exit FSCAN by entering

`QUit`

or by pressing the PF3 or PF15 key.

The FSCAN HELP Facility

FSCAN has a HELP facility. To use HELP, enter the command

Help

or press the PF1 or PF13 key. HELP displays a summary of FSCAN commands and prefix area commands, as shown in the sample screen below:

```

FSCAN FILE CAR      FOCUS A      HELP SCREEN 2 of 3

                                FSCAN COMMANDS
=          - Re-execute the most recent command line.
?          - Retrieve the previous command line.
Backward  - Go backward one screen.
CHange    - Change a string within a field:
           CHANGE fieldname=/oldstring/newstring/, $
CHILD     - Display child instances of this segment.
DElete    - Delete a segment instance, and all of its children.
DISplay   - Display the segment containing the specified fieldname.
End/FILE  - Save all changes and exit FSCAN.
FIND      - Find an instance on this chain which satisfies a test:
           FIND fieldname EQ GT CO... value.
FIRst     - Go to the first instance on this chain.
FORward   - Go forward one screen.
JumP     - Jump to the children of the next parent.
LAsT     - Go to the last instance on this chain
LEft     - Go left one panel.
LOcate   - Same as FIND but search is throughout the database.

Exit HELP: PF03/PF15. Forward: PF08/PF20. Backward: PF07/PF19.

```

You can scroll HELP screens back and forth by pressing the PF8 or PF20 key to go forward and the PF7 or PF19 key to go backward.

To exit the HELP facility, press the PF3 or PF15 key.

Syntax Summary

This section is a summary of the FSCAN commands, PF keys, and prefix area commands. References to other sections are included.

Summary of Commands

FSCAN commands are listed here in alphabetical order. The unique truncation of each command is capitalized.

Backward

Scrolls the display one screen backward.

PF keys: PF7 or PF19.

CHAnge

Changes character strings within field values. The syntax is

```
CHAnge field =/oldstring/newstring/ [,$ {*|n}]
```

where:

field

Is the name of the field whose value you want to change. The field must be alphanumeric and it cannot be a key field.

oldstring

Is the substring of the field value that you want to change.

newstring

Is the character string to replace the substring.

,\$ {|n}*

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the substring. If you want all instances in the group searched (starting from the current instance), use an asterisk (*).

You can also change field values by typing over them.

CHId

Displays the child instances relating to the current instance. (In SINGLE mode, displays the first child instance of the current instance.) The syntax is

`CHId [n]`

where:

n

Is the number of the child segment as assigned by FSCAN. If you omit this number, FSCAN displays a menu listing the segments and their numbers. Enter a number to display the segment (*Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 13-23).

Prefix area command: `C [n]`

where:

n

Is the number of the child segment as assigned by FSCAN. If you omit this number, FSCAN displays the menu.

DElete

Deletes the current instance and all descendant instances.

Prefix area command: `D`

Down [n]

Scrolls the display *n* lines forward. *n* defaults to 1.

Display Field Name

Displays the segment containing the specified field name.

End

Saves all changes made to the database and exits the FSCAN facility (see *Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands* on page 13-38).

FILE

Saves all changes made to the database and exits the FSCAN facility (see *Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands* on page 13-38).

FINd

Searches a group of instances (in the root segment, this is all instances in the segment; in descendant segments, this is all instances sharing a common parent instance) for an instance containing field values that fulfill certain conditions. FIND searches the group starting from the current instance. If it finds the instance, it makes that instance the current instance.

The syntax is (entered on one line)

```
FINd field1 rel1 value1 [OR value1a OR value1b OR ...]
```

```
[[AND|,} field2 rel2 value2 {AND|,}]
```

where:

fieldn ...

Is a field in the segment.

reln ...

Is one of the following relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance having the field value that passes the test becomes the current segment. If there are multiple tests, the first instance that passes all the tests becomes the current instance.

OR

Allows you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in the same FIND command.

First

Selects the first instance in a group of instances on display to be the current instance. In the root segment, the group of instances consists of all instances in the segment; in a descendant segment, a group consists of all instances that share a common parent instance.

Forward

Scrolls the display one screen forward.

PF keys: PF8 or PF20.

Help

Invokes the FSCAN HELP facility.

PF keys: PF01 or PF11.

Input

Adds a new segment instance.

Prefix area command: [I](#)

Note: This command is valid only in the input area as a prefix command.

Jump

Moves to the child of the next parent instance.

PF keys: PF12 or PF24

LAst

Selects the last instance of a group of instances on display. In the root segment, the group of instances consists of all instances in the segment; in a descendant segment, a group consists of all instances that share a common parent instance.

LEft

Scrolls the display one panel to the left.

PF keys: PF10 or PF22.

LOCate

Searches for instances containing field values that fulfill certain conditions. LOCATE searches starting from the current instance. If it finds the instance, it makes that instance the current instance.

The syntax is (entered on one line)

```
LOCate field1 rel1 value1 [OR value1a OR value1b OR ...]  
[{AND|,} field2 rel2 value2 {AND|,}]
```

where:

fieldn ...

Is a field to be tested.

reln ...

Is one of the following relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance having the field value that passes the test becomes the current segment. If there are multiple tests, the first instance that passes all the tests becomes the current instance.

OR

Allows you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in the same LOCATE command.

Key

Enables you to type over key field values in the current instance.

Prefix area command: `κ`

where:

`κ/`

Makes the instance the current instance after the key values are changed.

Multiple

Displays multiple instances, each on a single line. Entering this command after entering the SINGLE command returns the screen to the normal display (see *Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands* on page 13-26).

Next [n]

Scrolls the display *n* lines forward. *n* defaults to 1.

Parent

Displays the parent segment. The parent instance becomes the current instance. In SINGLE mode, PARENT displays the parent instance only.

QIt

Exits the FSCAN facility if you did not make any changes to the database.

PF keys: PF3 or PF15.

QQuit

Exits the FSCAN facility without saving any changes to the database.

REPlace

Replaces field values. The syntax is

```
REPlace field1 = value1[,field2 = value2 ...] [,$ {*|n}]
```

where:

fieldn...

Is a field in the instance whose value you want to change.

valuen...

Is the new value for the field.

,\$ {|n}*

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the field values you want to change. If you want all instances in the group searched (starting from the current instance), use an asterisk (*).

You can also replace field values by typing over them.

REPlace KEY

Replaces key field values in the current instance. The syntax is

```
REPlace KEY key1 = value1[, key2 = value2, ...]
```

where:

keyn ...

Is a key field in the instance whose value you want to change.

valuen ...

Is the new value for the key field.

You can also replace key field values by typing over them.

RESet

Performs the following:

- Clears the input area.
- Recovers all field values on the screen that you typed over, both non-key fields and key fields. To recover non-key field values, you must enter the RESET command before you press the Enter key. Otherwise, you will not recover the typed-over values.

PF keys: PF2 or PF14.

Prefix area command: **R**

Note: The R prefix-area command recovers only field values on the line that it is typed. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Right

Scrolls the display one panel to the right.

PF keys: PF11 or PF23.

SAve

Saves all changes made to the database without exiting FSCAN.

Single

Displays the current instance alone with all field values on one screen. To return to the normal display, enter the MULTIPLE command.

Top

Displays the root segment and makes the first instance in the root segment the current instance, scrolled to the leftmost panel.

?

Displays the previous command in stack.

PF keys: PF6 or PF18.

=

Executes the previous command entered.

PF key: PF9 or PF21.

Summary of PF Keys

The following table is a list of FSCAN PF keys and their corresponding functions.

FSCAN Keys	Functions
PF1, PF13	HELP
PF2, PF14	RESET
PF3, PF15	QUIT
PF4, PF16	PARENT
PF5, PF17	CHILD
PF6, PF18	?
PF7, PF19	BACKWARD
PF8, PF20	FORWARD
PF9, PF21	=
PF10, PF22	LEFT
PF11, PF23	RIGHT
PF12, PF24	JUMP

Summary of Prefix Area Commands

The following is a summary of prefix area commands. You type these commands in the prefix area that corresponds to the instance you wish to address.

/	Makes the instance the current instance. May be typed after the prefix area commands K, I, and R.
C	Displays child instances (see <i>Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands</i> on page 13-23).
D	Deletes the instance and all its children.
I	Inputs a new instance (valid only in the input area).
I/	Inputs a new instance and makes the instance the current instance (valid only in the input area).
K	Enables you to type over key field values in the instance.

K/	Enables you to type over key field values in the instance, then makes the instance the current instance.
R	<p>Performs the following:</p> <ul style="list-style-type: none">• Clears the input area.• Recovers all field values on the screen that you typed over, both non-key fields and key fields. To recover non-key field values, you must enter the RESET command before you press the Enter key. Otherwise, you will not recover the typed-over values. <p>Note that the R prefix area command recovers only field values on the line on which it is typed. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.</p>

APPENDIX A

Master Files and Diagrams

Topics:

- Creating Sample Data Sources
- EMPLOYEE Data Source
- JOBFIL Data Source
- EDUCFILE Data Source
- SALES Data Source
- PROD Data Source
- CAR Data Source
- LEDGER Data Source
- FINANCE Data Source
- REGION Data Source
- COURSES Data Source
- EMPDATA Data Source
- EXPERSON Data Source
- TRAINING Data Source
- PAYHIST File
- COMASTER File
- VIDEOTRK, MOVIES, and ITEMS Data Sources
- VIDEOTR2 Data Source
- Gotham Grinds Data Sources
- Century Corp Data Sources

This appendix contains descriptions and structure diagrams for the sample data sources used throughout the documentation.

Creating Sample Data Sources

Create sample data sources on your user ID by executing the procedures specified below. These FOCEXECs are supplied with FOCUS. If they are not available to you or if they produce error messages, contact your systems administrator.

To create these files, first make sure you have read access to the Master Files.

Data Source	Load Procedure Name
EMPLOYEE, EDUCFILE, and JOBFIL	<p>Under CMS, enter: EX EMPTEST</p> <p>Under MVS, enter: EX EMPTSO</p> <p>These FOCEXECs also test the data sources by generating sample reports. If you are using Hot Screen, remember to press either Enter or the PF3 key after each report. If the EMPLOYEE, EDUCFILE, and JOBFIL data sources already exist on your user ID, the FOCEXEC replaces them with new copies. This FOCEXEC assumes that the high-level qualifier for the FOCUS data sources is the same as the high-level qualifier for the MASTER PDS that was unloaded from the tape.</p>
SALES PROD	EX SALES EX PROD
CAR	None (created automatically during installation).
LEDGER FINANCE REGION COURSES EXPERSON	EX LEDGER EX FINANCE EX REGION EX COURSES EX EXPERSON
EMPDATA TRAINING	EX LOADEMP EX LOADTRAI
PAYHIST	None (PAYHIST DATA is a sequential data source and is allocated during the installation process).

Data Source	Load Procedure Name
COMASTER	None (COMASTER is used for debugging other Master Files).
VIDEOTRK and MOVIES	EX LOADVTRK
VIDEOTR2	EX LOADVID2
Gotham Grinds	EX LOADGG
Century Corp: CENTCOMP CENTFIN CENTHR CENTINV CENTORD CENTQA CENTGL CENTSYSF CENTSTMT	EX LOADCOM EX LOADFIN EX LOADHR EX LOADINV EX LOADORD EX LOADCQA EX LDCENTGL EX LDCENTSY EX LDSTMT

EMPLOYEE Data Source

EMPLOYEE contains sample data about a company's employees. Its segments are:

EMPINFO

Contains employee IDs, names, and positions.

FUNDTRAN

Specifies employees' direct deposit accounts. This segment is unique.

PAYINFO

Contains the employees' salary history.

ADDRESS

Contains employees' home and bank addresses.

SALINFO

Contains data on employees' monthly pay.

DEDUCT

Contains data on monthly pay deductions.

EMPLOYEE also contains cross-referenced segments belonging to the JOBFILE and EDUCFILE files, also described in this appendix. The segments are:

[JOBSEG](#) (from JOBFILE)

Describes the job positions held by each employee.

[SKILLSEG](#) (from JOBFILE)

Lists the skills required by each position.

[SECSEG](#) (from JOBFILE)

Specifies the security clearance needed for each job position.

[ATTNDSEG](#) (from EDUCFILE)

Lists the dates that employees attended in-house courses.

[COURSESEG](#) (from EDUCFILE)

Lists the courses that the employees attended.

EMPLOYEE Master File

```

FILENAME=EMPLOYEE, SUFFIX=FOC
SEGNAME=EMPINFO, SEGTYPE=S1
    FIELDNAME=EMP_ID, ALIAS=EID, FORMAT=A9, $
    FIELDNAME=LAST_NAME, ALIAS=LN, FORMAT=A15, $
    FIELDNAME=FIRST_NAME, ALIAS=FN, FORMAT=A10, $
    FIELDNAME=HIRE_DATE, ALIAS=HDT, FORMAT=I6YMD, $
    FIELDNAME=DEPARTMENT, ALIAS=DPT, FORMAT=A10, $
    FIELDNAME=CURR_SAL, ALIAS=CSAL, FORMAT=D12.2M, $
    FIELDNAME=CURR_JOBCODE, ALIAS=CJC, FORMAT=A3, $
    FIELDNAME=ED_HRS, ALIAS=OJT, FORMAT=F6.2, $
SEGNAME=FUNDTRAN, SEGTYPE=U, PARENT=EMPINFO
    FIELDNAME=BANK_NAME, ALIAS=BN, FORMAT=A20, $
    FIELDNAME=BANK_CODE, ALIAS=BC, FORMAT=I6S, $
    FIELDNAME=BANK_ACCT, ALIAS=BA, FORMAT=I9S, $
    FIELDNAME=EFFECT_DATE, ALIAS=EDATE, FORMAT=I6YMD, $
SEGNAME=PAYINFO, SEGTYPE=SH1, PARENT=EMPINFO
    FIELDNAME=DAT_INC, ALIAS=DI, FORMAT=I6YMD, $
    FIELDNAME=PCT_INC, ALIAS=PI, FORMAT=F6.2, $
    FIELDNAME=SALARY, ALIAS=SAL, FORMAT=D12.2M, $
    FIELDNAME=JOBCODE, ALIAS=JBC, FORMAT=A3, $
SEGNAME=ADDRESS, SEGTYPE=S1, PARENT=EMPINFO
    FIELDNAME=TYPE, ALIAS=AT, FORMAT=A4, $
    FIELDNAME=ADDRESS_LN1, ALIAS=LN1, FORMAT=A20, $
    FIELDNAME=ADDRESS_LN2, ALIAS=LN2, FORMAT=A20, $
    FIELDNAME=ADDRESS_LN3, ALIAS=LN3, FORMAT=A20, $
    FIELDNAME=ACCTNUMBER, ALIAS=ANO, FORMAT=I9L, $
SEGNAME=SALINFO, SEGTYPE=SH1, PARENT=EMPINFO
    FIELDNAME=PAY_DATE, ALIAS=PD, FORMAT=I6YMD, $
    FIELDNAME=GROSS, ALIAS=MO_PAY, FORMAT=D12.2M, $
SEGNAME=DEDUCT, SEGTYPE=S1, PARENT=SALINFO
    FIELDNAME=DED_CODE, ALIAS=DC, FORMAT=A4, $
    FIELDNAME=DED_AMT, ALIAS=DA, FORMAT=D12.2M, $
SEGNAME=JOBSEG, SEGTYPE=KU, PARENT=PAYINFO, CRFILE=JOBFILE,
    CRKEY=JOBCODE, $
SEGNAME=SECSEG, SEGTYPE=KLU, PARENT=JOBSEG, CRFILE=JOBFILE, $
SEGNAME=SKILLSEG, SEGTYPE=KL, PARENT=JOBSEG, CRFILE=JOBFILE, $
SEGNAME=ATTNDSEG, SEGTYPE=KM, PARENT=EMPINFO, CRFILE=EDUCFILE,
    CRKEY=EMP_ID, $
SEGNAME=COURSEG, SEGTYPE=KLU, PARENT=ATTNDSEG, CRFILE=EDUCFILE, $

```


JOBFILE Data Source

JOBFILE contains sample data about a company's job positions. Its segments are:

JOBSEG

Describes what each position is. The field JOBCODE in this segment is indexed.

SKILLSEG

Lists the skills required by each position.

SECSEG

Specifies the security clearance needed, if any. This segment is unique.

JOBFILE Master File

```

FILENAME=JOBFILE,  SUFFIX=FOC
SEGNAME=JOBSEG,   SEGTYPE=S1
  FIELDNAME=JOBCODE,  ALIAS=JC,  FORMAT=A3,    INDEX=I,$
  FIELDNAME=JOB_DESC, ALIAS=JD,  FORMAT=A25,    , $
SEGNAME=SKILLSEG,  SEGTYPE=S1,  PARENT=JOBSEG
  FIELDNAME=SKILLS,  ALIAS=,    FORMAT=A4,    , $
  FIELDNAME=SKILL_DESC, ALIAS=SD,  FORMAT=A30,   , $
SEGNAME=SECSEG,   SEGTYPE=U,   PARENT=JOBSEG
  FIELDNAME=SEC_CLEAR, ALIAS=SC,  FORMAT=A6,    , $

```

JOBFILE Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE JOBFILE ON 05/15/03 AT 14.40.06

```

          JOBSEG
01          S1
*****
*JOBCODE   **I
*JOB_DESC  **
*          **
*          **
*          **
*****
          I
          +-----+
          I          I
          I SECSEG   I SKILLSEG
02          I U          03          I S1
*****          *****
*SEC_CLEAR *          *SKILLS   **
*          *          *SKILL_DESC **
*          *          *          **
*          *          *          **
*          *          *          **
*****          *****
                   *****

```

EDUCFILE Data Source

EDUCFILE contains sample data about a company's in-house courses. Its segments are:

COURSESEG

Contains data on each course.

ATTNDSEG

Specifies which employees attended the courses. Both fields in the segment are key fields. The field EMP_ID in this segment is indexed.

EDUCFILE Master File

```
FILENAME=EDUCFILE, SUFFIX=FOC
SEGNAME=COURSESEG, SEGTYPE=S1
  FIELDNAME=COURSE_CODE, ALIAS=CC, FORMAT=A6, $
  FIELDNAME=COURSE_NAME, ALIAS=CD, FORMAT=A30, $
SEGNAME=ATTNDSEG, SEGTYPE=SH2, PARENT=COURSESEG
  FIELDNAME=DATE_ATTEND, ALIAS=DA, FORMAT=I6YMD, $
  FIELDNAME=EMP_ID, ALIAS=EID, FORMAT=A9, INDEX=I, $
```

EDUCFILE Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE EDUCFILE ON 05/15/03 AT 14.45.44

```
          COURSESEG
01          S1
*****
*COURSE_CODE **
*COURSE_NAME **
*           **
*           **
*           **
*****
          I
          I
          I
          I ATTNDSEG
02          I SH2
*****
*DATE_ATTEND **
*EMP_ID       **I
*           **
*           **
*           **
*****
          I
          I
          I
```

SALES Data Source

SALES contains sample data about a dairy company with an affiliated store chain. Its segments are:

STOR_SEG

Lists the stores buying the products.

DAT_SEG

Contains the dates of inventory.

PRODUCT

Contains sales data for each product on each date. The PROD_CODE field is indexed. The RETURNS and DAMAGED fields have the MISSING=ON attribute.

SALES Master File

```

FILENAME=KSALES,   SUFFIX=FOC
SEGNAME=STOR_SEG, SEGTYPE=S1
  FIELDNAME=STORE_CODE, ALIAS=SNO,   FORMAT=A3,   $
  FIELDNAME=CITY,       ALIAS=CTY,   FORMAT=A15,  $
  FIELDNAME=AREA,       ALIAS=LOC,   FORMAT=A1,   $
SEGNAME=DATE_SEG, PARENT=STOR_SEG, SEGTYPE=SH1,
  FIELDNAME=DATE,       ALIAS=DTE,   FORMAT=A4MD, $
SEGNAME=PRODUCT, PARENT=DATE_SEG, SEGTYPE=S1,
  FIELDNAME=PROD_CODE, ALIAS=PCODE,  FORMAT=A3,   FIELDTYPE=I,$
  FIELDNAME=UNIT_SOLD, ALIAS=SOLD,   FORMAT=I5,   $
  FIELDNAME=RETAIL_PRICE,ALIAS=RP,   FORMAT=D5.2M,$
  FIELDNAME=DELIVER_AMT, ALIAS=SHIP,  FORMAT=I5,   $
  FIELDNAME=OPENING_AMT, ALIAS=INV,   FORMAT=I5,   $
  FIELDNAME=RETURNS,     ALIAS=RTN,   FORMAT=I3,   MISSING=ON,$
  FIELDNAME=DAMAGED,     ALIAS=BAD,   FORMAT=I3,   MISSING=ON,$

```

SALES Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE SALES ON 05/15/03 AT 14.50.28

```
          STOR_SEG
01          S1
*****
*STORE_CODE **
*CITY        **
*AREA        **
*            **
*            **
*****
          I
          I
          I
          I DATE_SEG
02          I SH1
*****
*DATE        **
*            **
*            **
*            **
*            **
*****
          I
          I
          I
          I PRODUCT
03          I S1
*****
*PROD_CODE  **I
*UNIT_SOLD  **
*RETAIL_PRICE**
*DELIVER_AMT **
*            **
*****
          *****
```


PROD Data Source

The PROD data source lists products sold by a dairy company. It consists of one segment, PRODUCT. The field PROD_CODE is indexed.

PROD Master File

```
FILE=KPROD, SUFFIX=FOC,
SEGMENT=PRODUCT, SEGTYPE=S1,
  FIELDNAME=PROD_CODE, ALIAS=PCODE, FORMAT=A3,   FIELDTYPE=I, $
  FIELDNAME=PROD_NAME, ALIAS=ITEM,  FORMAT=A15,   $
  FIELDNAME=PACKAGE,   ALIAS=SIZE,   FORMAT=A12,   $
  FIELDNAME=UNIT_COST, ALIAS=COST,   FORMAT=D5.2M, $
```

PROD Structure Diagram

```
SECTION 01
          STRUCTURE OF FOCUS   FILE PROD   ON 05/15/03 AT 14.57.38
          PRODUCT
01          S1
*****
*PROD_CODE   ** I
*PROD_NAME   **
*PACKAGE     **
*UNIT_COST   **
*            **
*****
*****
```

CAR Data Source

CAR contains sample data about specifications and sales information for rare cars. Its segments are:

ORIGIN

Lists the country that manufactures the car. The field COUNTRY is indexed.

COMP

Contains the car name.

CARREC

Contains the car model.

BODY

Lists the body type, seats, dealer and retail costs, and units sold.

SPECS

Lists car specifications. This segment is unique.

WARANT

Lists the type of warranty.

EQUIP

Lists standard equipment.

The aliases in the CAR Master File are specified without the ALIAS keyword.

CAR Master File

```

FILENAME=CAR, SUFFIX=FOC
SEGNAME=ORIGIN, SEGTYPE=S1
  FIELDNAME=COUNTRY, COUNTRY, A10, FIELDTYPE=I, $
SEGNAME=COMP, SEGTYPE=S1, PARENT=ORIGIN
  FIELDNAME=CAR, CARS, A16, $
SEGNAME=CARREC, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=MODEL, MODEL, A24, $
SEGNAME=BODY, SEGTYPE=S1, PARENT=CARREC
  FIELDNAME=BODYTYPE, TYPE, A12, $
  FIELDNAME=SEATS, SEAT, I3, $
  FIELDNAME=DEALER_COST, DCOST, D7, $
  FIELDNAME=RETAIL_COST, RCOST, D7, $
  FIELDNAME=SALES, UNITS, I6, $
SEGNAME=SPECS, SEGTYPE=U, PARENT=BODY
  FIELDNAME=LENGTH, LEN, D5, $
  FIELDNAME=WIDTH, WIDTH, D5, $
  FIELDNAME=HEIGHT, HEIGHT, D5, $
  FIELDNAME=WEIGHT, WEIGHT, D6, $
  FIELDNAME=WHEELBASE, BASE, D6.1, $
  FIELDNAME=FUEL_CAP, FUEL, D6.1, $
  FIELDNAME=BHP, POWER, D6, $
  FIELDNAME=RPM, RPM, I5, $
  FIELDNAME=MPG, MILES, D6, $
  FIELDNAME=ACCEL, SECONDS, D6, $
SEGNAME=WARRANT, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=WARRANTY, WARR, A40, $
SEGNAME=EQUIP, SEGTYPE=S1, PARENT=COMP
  FIELDNAME=STANDARD, EQUIP, A40, $

```

CAR Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE CAR

ON 05/15/03 AT 10.16.27

```

      ORIGIN
01      S1
*****
*COUNTRY      **I
*              **
*              **
*              **
*              **
*****
      I
      I
      I
      I COMP
02      I S1
*****
*CAR          **
*              **
*              **
*              **
*              **
*****
      I
      +-----+-----+
      I              I              I
      I CARREC      I WARRANT      I EQUIP
03      I S1        06      I S1      07      I S1
*****
*MODEL        *      *WARRANTY    **      *STANDARD    **
*              *      *              **      *              **
*              *      *              **      *              **
*              *      *              **      *              **
*              *      *              **      *              **
*****
*****
*****
      I
      I
      I
      I BODY
04      I S1
*****
*BODYPYPE    **
*SEATS       **
*DEALER_COST **
*RETAIL_COST **
*              **
*****
*****
      I
      I
      I
      I SPECS
05      I U
*****
*LENGTH      **
*WIDTH       **
*HEIGHT      **
*WEIGHTT     **
*              **
*****
*****
```

LEDGER Data Source

LEDGER contains sample accounting data. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

LEDGER Master File

```

FILENAME=LEDGER, SUFFIX=FOC,$
SEGNAME=TOP, SEGTYPE=S2,$
FIELDNAME=YEAR , , FORMAT=A4, $
FIELDNAME=ACCOUNT, , FORMAT=A4, $
FIELDNAME=AMOUNT , , FORMAT=I5C,$
    
```

LEDGER Structure Diagram

```

SECTION 01
                STRUCTURE OF FOCUS   FILE LEDGER   ON 05/15/03 AT 15.17.08

                TOP
01             S2
*****
*YEAR          **
*ACCOUNT       **
*AMOUNT        **
*              **
*              **
*****
*****
    
```

FINANCE Data Source

FINANCE contains sample financial data for balance sheets. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

FINANCE Master File

```
FILENAME=FINANCE, SUFFIX=FOC,$
SEGNAME=TOP, SEGTYPE=S2,$
FIELDNAME=YEAR , , FORMAT=A4, $
FIELDNAME=ACCOUNT, , FORMAT=A4, $
FIELDNAME=AMOUNT , , FORMAT=D12C,$
```

FINANCE Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE FINANCE ON 05/15/03 AT 15.17.08

```
TOP
01 S2
*****
*YEAR **
*ACCOUNT **
*AMOUNT **
* **
* **
*****
*****
```

REGION Data Source

REGION contains sample account data for the eastern and western regions of the country. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

REGION Master File

```
FILENAME=REGION, SUFFIX=FOC,$
SEGNAME=TOP, SEGTYPE=S1,$
FIELDNAME=ACCOUNT, , FORMAT=A4, $
FIELDNAME=E_ACTUAL, , FORMAT=I5C,$
FIELDNAME=E_BUDGET, , FORMAT=I5C,$
FIELDNAME=W_ACTUAL, , FORMAT=I5C,$
FIELDNAME=W_BUDGET, , FORMAT=I5C,$
```

REGION Structure Diagram

```
SECTION 01
```

```
STRUCTURE OF FOCUS FILE REGION ON 05/15/03 AT 15.18.48
```

```
TOP
01 S1
*****
*ACCOUNT **
*E_ACTUAL **
*E_BUDGET **
*W_ACTUAL **
* **
*****
*****
```

COURSES Data Source

COURSES contains sample data about education courses. It consists of one segment, CRSESEG1. The field DESCRIPTION has a format of TEXT (TX).

COURSES Master File

```
FILENAME=COURSES,   SUFFIX=FOC,                               $
SEGNAME=CRSESEG1,  SEGTYPE=S1,                               $
  FIELDNAME=COURSE_CODE,  ALIAS=CC,    FORMAT=A6,    FIELDTYPE=I,  $
  FIELDNAME=COURSE_NAME,  ALIAS=CN,    FORMAT=A30,   $
  FIELDNAME=DURATION,     ALIAS=DAYS,   FORMAT=I3,    $
  FIELDNAME=DESCRIPTION,  ALIAS=CDESC,  FORMAT=TX50,  $
```

COURSES Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE COURSES ON 05/15/03 AT 12.26.05

```
          CRSESEG1
01      S1
*****
*COURSE_CODE **I
*COURSE_NAME **
*DURATION    **
*DESCRIPTION **T
*           **
*****
*****
```


EMPDATA Data Source

EMPDATA contains sample data about a company's employees. It consists of one segment, EMPDATA. The PIN field is indexed. The AREA field is a temporary field.

EMPDATA Master File

```

FILENAME=EMPDATA, SUFFIX=FOC
SEGNAME=EMPDATA, SEGTYPE=S1
  FIELDNAME=PIN,           ALIAS=ID,           FORMAT=A9,   INDEX=I,   $
  FIELDNAME=LASTNAME,     ALIAS=LN,           FORMAT=A15,   $
  FIELDNAME=FIRSTNAME,   ALIAS=FN,           FORMAT=A10,   $
  FIELDNAME=MIDINITIAL,  ALIAS=MI,           FORMAT=A1,    $
  FIELDNAME=DIV,         ALIAS=CDIV,         FORMAT=A4,    $
  FIELDNAME=DEPT,        ALIAS=CDEPT,        FORMAT=A20,   $
  FIELDNAME=JOBCLASS,    ALIAS=CJCLAS,       FORMAT=A8,    $
  FIELDNAME=TITLE,       ALIAS=CFUNC,        FORMAT=A20,   $
  FIELDNAME=SALARY,      ALIAS=CSAL,         FORMAT=D12.2M,$
  FIELDNAME=HIREDATE,    ALIAS=HDAT,         FORMAT=YMD,   $
$
DEFINE AREA/A13=DECODE DIV (NE 'NORTH EASTERN' SE 'SOUTH EASTERN'
CE 'CENTRAL' WE 'WESTERN' CORP 'CORPORATE' ELSE 'INVALID AREA');$

```

EMPDATA Structure Diagram

```

SECTION 01
          STRUCTURE OF FOCUS      FILE EMPDATA   ON 05/15/03 AT 14.49.09

          EMPDATA
01          S1
*****
*PIN          **I
*LASTNAME     **
*FIRSTNAME    **
*MIDINITIAL   **
*             **
*****
*****

```

EXPERSON Data Source

The EXPERSON data source contains personal data about individual employees. It consists of one segment, ONESEG.

EXPERSON Master File

```

FILE=EXPERSON      , SUFFIX=FOC
SEGMENT=ONESEG, $
  FIELDNAME=SOC_SEC_NO      , ALIAS=SSN          , USAGE=A9      , $
  FIELDNAME=FIRST_NAME     , ALIAS=FN           , USAGE=A9      , $
  FIELDNAME=LAST_NAME      , ALIAS=LN           , USAGE=A10     , $
  FIELDNAME=AGE             , ALIAS=YEAR        , USAGE=I2      , $
  FIELDNAME=SEX             , ALIAS=             , USAGE=A1      , $
  FIELDNAME=MARITAL_STAT   , ALIAS=MS          , USAGE=A1      , $
  FIELDNAME=NO_DEP         , ALIAS=NDP         , USAGE=I3      , $
  FIELDNAME=DEGREE         , ALIAS=            , USAGE=A3      , $
  FIELDNAME=NO_CARS        , ALIAS=CARS        , USAGE=I3      , $
  FIELDNAME=ADDRESS        , ALIAS=            , USAGE=A14     , $
  FIELDNAME=CITY           , ALIAS=            , USAGE=A10     , $
  FIELDNAME=WAGE           , ALIAS=PAY         , USAGE=D10.2SM , $
  FIELDNAME=CATEGORY       , ALIAS=STATUS      , USAGE=A1      , $
  FIELDNAME=SKILL_CODE     , ALIAS=SKILLS      , USAGE=A5      , $
  FIELDNAME=DEPT_CODE      , ALIAS=WHERE       , USAGE=A4      , $
  FIELDNAME=TEL_EXT        , ALIAS=EXT         , USAGE=I4      , $
  FIELDNAME=DATE_EMP       , ALIAS=BASE_DATE   , USAGE=I6YMTD  , $
  FIELDNAME=MULTIPLIER     , ALIAS=RATIO       , USAGE=D5.3    , $
  
```

EXPERSON Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE EXPERSON ON 05/15/03 AT 14.50.58

```

      ONESEG
01      S1
*****
*SOC_SEC_NO  **
*FIRST_NAME **
*LAST_NAME  **
*AGE        **
*           **
*****
*****
  
```

TRAINING Data Source

TRAINING contains sample data about training courses for employees. It consists of one segment, TRAINING. The PIN field is indexed. The EXPENSES, GRADE, and LOCATION fields have the MISSING=ON attribute.

TRAINING Master File

```

FILENAME=TRAINING, SUFFIX=FOC
SEGNAME=TRAINING, SEGTYPE=SH3
  FIELDNAME=PIN,           ALIAS=ID,           FORMAT=A9,     INDEX=I,      $
  FIELDNAME=COURSESTART,  ALIAS=CSTART,  FORMAT=YMD,    $
  FIELDNAME=COURSECODE,  ALIAS=CCOD,    FORMAT=A7,     $
  FIELDNAME=EXPENSES,    ALIAS=COST,    FORMAT=D8.2,   MISSING=ON    $
  FIELDNAME=GRADE,       ALIAS=GRA,     FORMAT=A2,     MISSING=ON,   $
  FIELDNAME=LOCATION,     ALIAS=LOC,     FORMAT=A6,     MISSING=ON,   $

```

TRAINING Structure Diagram

```
SECTION 01
```

```
STRUCTURE OF FOCUS FILE TRAINING ON 05/15/03 AT 14.51.28
```

```

          TRAINING
01      SH3
*****
*PIN           **I
*COURSESTART  **
*COURSECODE   **
*EXPENSES     **
*             **
*****
*****

```

PAYHIST File

The PAYHIST data source contains the employees' salary history. It consists of one segment, PAYSEG. The SUFFIX attribute indicates that the data file is a fixed-format sequential file.

PAYHIST Master File

```

FILENAME=PAYHIST,  SUFFIX=FIX
SEGMENT=PAYSEG, $
  FIELDNAME=SOC_SEC_NO,  ALIAS=SSN,          USAGE=A9,          ACTUAL=A9,  $
  FIELDNAME=DATE_OF_IN,  ALIAS=INCDATE,      USAGE=I6YMTD,     ACTUAL=A6,  $
  FIELDNAME=AMT_OF_INC,  ALIAS=RAISE,        USAGE=D6.2,       ACTUAL=A10, $
  FIELDNAME=PCT_INC,     ALIAS=,            USAGE=D6.2,       ACTUAL=A6,  $
  FIELDNAME=NEW_SAL,     ALIAS=CURR_SAL,    USAGE=D10.2,     ACTUAL=A11, $
  FIELDNAME=FILL,        ALIAS=,            USAGE=A38,       ACTUAL=A38, $
  
```

PAYHIST Structure Diagram

```

SECTION 01
          STRUCTURE OF FIX      FILE PAYHIST ON 05/15/03 AT 14.51.59
          PAYSEG
01      S1
*****
*SOC_SEC_NO  **
*DATE_OF_IN  **
*AMT_OF_INC  **
*PCT_INC     **
*           **
*****
*****
  
```

COMASTER File

The COMASTER file is used to display the file structure and contents of each segment in a data source. Since COMASTER is used for debugging other Master Files, a corresponding FOCEXEC does not exist for the COMASTER file. Its segments are:

- FILEID, which lists file information.
- RECID, which lists segment information.
- FIELDID, which lists field information.
- DEFREC, which lists a description record.
- PASSREC, which lists read/write access.
- CRSEG, which lists cross-reference information for segments.
- ACCSEG, which lists DBA information.

COMASTER Master File

```

SUFFIX=COM, SEGNAME=FILEID
  FIELDNAME=FILENAME      , FILE          , A8 ,      , $
  FIELDNAME=FILE SUFFIX  , SUFFIX        , A8 ,      , $
  FIELDNAME=FDFCENT      , FDFC          , A4 ,      , $
  FIELDNAME=FYRTHRESH   , FYRT          , A2 ,      , $
SEGNAME=RECID
  FIELDNAME=SEGNAME      , SEGMENT       , A8 ,      , $
  FIELDNAME=SEGTYPE      , SEGTYPE       , A4 ,      , $
  FIELDNAME=SEGSIZE      , SEGSIZE       , I4 ,      A4 , $
  FIELDNAME=PARENT       , PARENT        , A8 ,      , $
  FIELDNAME=CRKEY        , VKEY          , A66 ,     , $
SEGNAME=FIELDID
  FIELDNAME=FIELDNAME    , FIELD         , A66 ,     , $
  FIELDNAME=ALIAS        , SYNONYM       , A66 ,     , $
  FIELDNAME=FORMAT       , USAGE         , A8 ,      , $
  FIELDNAME=ACTUAL       , ACTUAL        , A8 ,      , $
  FIELDNAME=AUTHORITY    , AUTHCODE      , A8 ,      , $
  FIELDNAME=FIELDTYPE    , INDEX         , A8 ,      , $
  FIELDNAME=TITLE        , TITLE         , A64 ,     , $
  FIELDNAME=HELPMESSAGE  , MESSAGE       , A256 ,    , $
  FIELDNAME=MISSING      , MISSING       , A4 ,      , $
  FIELDNAME=ACCEPTS      , ACCEPTABLE    , A255 ,    , $
  FIELDNAME=RESERVED     , RESERVED      , A44 ,     , $
  FIELDNAME=DFCENT       , DFC           , A4 ,      , $
  FIELDNAME=YRTHRESH     , YRT           , A4 ,      , $
SEGNAME=DEFREC
  FIELDNAME=DEFINITION   , DESCRIPTION    , A44 ,     , $
SEGNAME=PASSREC, PARENT=FILEID
  FIELDNAME=READ/WRITE   , RW            , A32 ,     , $
SEGNAME=CRSEG, PARENT=RECID
  FIELDNAME=CRFILENAME   , CRFILE        , A8 ,      , $
  FIELDNAME=CRSEGNAME    , CRSEGMENT     , A8 ,      , $
  FIELDNAME=ENCRYPT       , ENCRYPT        , A4 ,      , $
SEGNAME=ACCSEG, PARENT=DEFREC
  FIELDNAME=DBA          , DBA           , A8 ,      , $
  FIELDNAME=DBAFILE      ,               , A8 ,      , $
  FIELDNAME=USER         , PASS          , A8 ,      , $
  FIELDNAME=ACCESS       , ACCESS        , A8 ,      , $
  FIELDNAME=RESTRICT     , RESTRICT      , A8 ,      , $
  FIELDNAME=NAME         , NAME          , A66 ,     , $
  FIELDNAME=VALUE        , VALUE         , A80 ,     , $

```

COMASTER Structure Diagram

```

SECTION 01
      STRUCTURE OF EXTERNAL FILE COMASTER ON 05/15/03 AT 14.53.38
      FILEID
01      S1
*****
*FILENAME      **
*FILE SUFFIX  **
*FDFCENT      **
*PYRTHRESH   **
*              **
*****
      I
      +-----+
      I              I
      I RECID        I PASSREC
02      I N          07      I N
*****
*SEGNAME      **      *READ/WRITE **
*SEGTYPE      **      *              **
*SEGSIZE      **      *              **
*PARENT       **      *              **
*              **      *              **
*****
*              **      *              **
*****
      I
      +-----+
      I              I
      I FIELDID      I CRSEG
03      I N          06      I N
*****
*FIELDNAME    **      *CRFILENAME **
*ALIAS        **      *CRSEGMNAME **
*FORMAT       **      *ENCRYPT     **
*ACTUAL       **      *              **
*              **      *              **
*****
*              **      *              **
*****
      I
      I
      I
      I DEFREC
04      I N
*****
*DEFINITION   **
*              **
*              **
*              **
*              **
*****
*              **
*****
      I
      I
      I
      I ACCSEG
05      I N
*****
*DBA          **
*DBAFILE      **
*USER         **
*ACCESS       **
*              **
*****
*              **
*****

```

VIDEOTRK, MOVIES, and ITEMS Data Sources

VIDEOTRK contains sample data about customer, rental, and purchase information for a video rental business. It can be joined to the MOVIES or ITEMS data source. VIDEOTRK and MOVIES are used in examples that illustrate the use of the Maintain facility.

VIDEOTRK Master File

```

FILENAME=VIDEOTRK, SUFFIX=FOC
SEGNAME=CUST, SEGTYPE=S1
  FIELDNAME=CUSTID, ALIAS=CIN, FORMAT=A4, $
  FIELDNAME=LASTNAME, ALIAS=LN, FORMAT=A15, $
  FIELDNAME=FIRSTNAME, ALIAS=FN, FORMAT=A10, $
  FIELDNAME=EXPDATE, ALIAS=EXDAT, FORMAT=YMD, $
  FIELDNAME=PHONE, ALIAS=TEL, FORMAT=A10, $
  FIELDNAME=STREET, ALIAS=STR, FORMAT=A20, $
  FIELDNAME=CITY, ALIAS=CITY, FORMAT=A20, $
  FIELDNAME=STATE, ALIAS=PROV, FORMAT=A4, $
  FIELDNAME=ZIP, ALIAS=POSTAL_CODE, FORMAT=A9, $
SEGNAME=TRANSDAT, SEGTYPE=SH1, PARENT=CUST
  FIELDNAME=TRANSDATE, ALIAS=OUTDATE, FORMAT=YMD, $
SEGNAME=SALES, SEGTYPE=S2, PARENT=TRANSDAT
  FIELDNAME=PRODCODE, ALIAS=PCOD, FORMAT=A6, $
  FIELDNAME=TRANSCODE, ALIAS=TCOD, FORMAT=I3, $
  FIELDNAME=QUANTITY, ALIAS=NO, FORMAT=I3S, $
  FIELDNAME=TRANSTOT, ALIAS=TTOT, FORMAT=F7.2S, $
SEGNAME=RENTALS, SEGTYPE=S2, PARENT=TRANSDAT
  FIELDNAME=MOVIECODE, ALIAS=MCOD, FORMAT=A6, INDEX=I, $
  FIELDNAME=COPY, ALIAS=COPY, FORMAT=I2, $
  FIELDNAME=RETURNDATE, ALIAS=INDATE, FORMAT=YMD, $
  FIELDNAME=FEE, ALIAS=FEE, FORMAT=F5.2S, $

```

VIDEOTRK Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE VIDEOTRK ON 05/15/03 AT 12.25.19

```

          CUST
01      S1
*****
*CUSTID      **
*LASTNAME   **
*FIRSTNAME  **
*EXPDATE    **
*           **
*****
          I
          I
          I
          I  TRANSDAT
02      I  SH1
*****
*TRANSDATE  **
*           **
*           **
*           **
*           **
*****
          I
          +-----+
          I           I
          I  SALES    I  RENTALS
03      I  S2          04      I  S2
*****                *****
*PRODCODE    **      *MOVIECODE  ** I
*TRANSCODE   **      *COPY        **
*QUANTITY    **      *RETURNDATE **
*TRANSTOT    **      *FEE         **
*           **      *           **
*****                *****
          *****                *****
          *****                *****

```


MOVIES Master File

```

FILENAME=MOVIES,      SUFFIX=FOC
SEGNAME=MOVINFO,     SEGTYPE=S1
  FIELDNAME=MOVIECODE,  ALIAS=MCOD,  FORMAT=A6, INDEX=I, $
  FIELDNAME=TITLE,     ALIAS=MTL,   FORMAT=A39,  $
  FIELDNAME=CATEGORY,  ALIAS=CLASS, FORMAT=A8,   $
  FIELDNAME=DIRECTOR,  ALIAS=DIR,   FORMAT=A17,  $
  FIELDNAME=RATING,    ALIAS=RTG,   FORMAT=A4,   $
  FIELDNAME=RELDATE,   ALIAS=RDAT,  FORMAT=YMD,  $
  FIELDNAME=WHOLESALEPR, ALIAS=WPRC,  FORMAT=F6.2, $
  FIELDNAME=LISTPR,    ALIAS=LPRC,  FORMAT=F6.2, $
  FIELDNAME=COPIES,    ALIAS=NOC,   FORMAT=I3,   $
  
```

MOVIES Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE MOVIES ON 05/15/03 AT 12.26.05

```

          MOVINFO
01      S1
*****
*MOVIECODE  **I
*TITLE      **
*CATEGORY   **
*DIRECTOR   **
*           **
*****
*****
  
```

ITEMS Master File

```
FILENAME=ITEMS,      SUFFIX=FOC
SEGNAME=ITMINFO,    SEGTYPE=S1
  FIELDNAME=PRODCODE, ALIAS=PCOD,  FORMAT=A6,  INDEX=I,  $
  FIELDNAME=PRODNAME, ALIAS=PROD,  FORMAT=A20,          $
  FIELDNAME=OURCOST,   ALIAS=WCost,  FORMAT=F6.2,         $
  FIELDNAME=RETAILPR, ALIAS=PRICE,  FORMAT=F6.2,         $
  FIELDNAME=ON_HAND,  ALIAS=NUM,    FORMAT=I5,           $
```

ITEMS Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE ITEMS ON 05/15/03 AT 12.26.05

```
          ITMINFO
01      S1
*****
*PRODCODE   **I
*PRODNAME   **
*OURCOST    **
*RETAILPR   **
*           **
*****
*****
```

VIDEOTR2 Data Source

VIDEOTR2 contains sample data about customer, rental, and purchase information for a video rental business. It consists of four segments.

VIDEOTR2 Master File

```

FILENAME=VIDEOTR2, SUFFIX=FOC
SEGNAME=CUST, SEGTYPE=S1
  FIELDNAME=CUSTID, ALIAS=CIN, FORMAT=A4, $
  FIELDNAME=LASTNAME, ALIAS=LN, FORMAT=A15, $
  FIELDNAME=FIRSTNAME, ALIAS=FN, FORMAT=A10, $
  FIELDNAME=EXPDATE, ALIAS=EXDAT, FORMAT=YMD, $
  FIELDNAME=PHONE, ALIAS=TEL, FORMAT=A10, $
  FIELDNAME=STREET, ALIAS=STR, FORMAT=A20, $
  FIELDNAME=CITY, ALIAS=CITY, FORMAT=A20, $
  FIELDNAME=STATE, ALIAS=PROV, FORMAT=A4, $
  FIELDNAME=ZIP, ALIAS=POSTAL_CODE, FORMAT=A9, $
SEGNAME=TRANSDAT, SEGTYPE=SH1, PARENT=CUST
  FIELDNAME=TRANSDATE, ALIAS=OUTDATE, FORMAT=HYMDI, $
SEGNAME=SALES, SEGTYPE=S2, PARENT=TRANSDAT
  FIELDNAME=TRANSCODE, ALIAS=TCOD, FORMAT=I3, $
  FIELDNAME=QUANTITY, ALIAS=NO, FORMAT=I3S, $
  FIELDNAME=TRANSTOT, ALIAS=TTOT, FORMAT=F7.2S, $
SEGNAME=RENTALS, SEGTYPE=S2, PARENT=TRANSDAT
  FIELDNAME=MOVIECODE, ALIAS=MCOD, FORMAT=A6, INDEX=I, $
  FIELDNAME=COPY, ALIAS=COPY, FORMAT=I2, $
  FIELDNAME=RETURNDATE, ALIAS=INDATE, FORMAT=YMD, $
  FIELDNAME=FEE, ALIAS=FEE, FORMAT=F5.2S, $

```

VIDEOTR2 Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE VIDEOTR2 ON 05/15/03 AT 16.45.48

```

          CUST
01      S1
*****
*CUSTID      **
*LASTNAME    **
*FIRSTNAME   **
*EXPDATE     **
*            **
*****
          I
          I
          I
          I TRANSDAT
02      I SH1
*****
*TRANSDATE   **
*            **
*            **
*            **
*            **
*****
          I
          +-----+
          I              I
          I SALES        I RENTALS
03      I S2            04      I S2
*****                *****
*TRANSCODE   **      *MOVIECODE   ** I
*QUANTITY    **      *COPY         **
*TRANSTOT    **      *RETURNDATE  **
*            **      *FEE          **
*            **      *            **
*****                *****
          *****                *****

```

Gotham Grinds Data Sources

Gotham Grinds is a group of data sources that contain sample data about a specialty items company.

- GGDEMOG contains demographic information about the customers of Gotham Grinds, a company that sells specialty items like coffee, gourmet snacks, and gifts. It consists of one segment, DEMOG01.
- GGORDER contains order information for Gotham Grinds. It consists of two segments, ORDER01 and ORDER02.
- GGPRODS contains product information for Gotham Grinds. It consists of one segment, PRODS01.
- GGSALES contains sales information for Gotham Grinds. It consists of one segment, SALES01.
- GGSTORES contains information for each of Gotham Grinds' 12 stores in the United States. It consists of one segment, STORES01.

GGDEMOG Master File

```

FILENAME=GGDEMOG, SUFFIX=FOC
SEGNAME=DEMOG01, SEGTYPE=S1
  FIELD=ST, ALIAS=E02, FORMAT=A02, INDEX=I, TITLE='State',
  DESC='State', $
  FIELD=HH, ALIAS=E03, FORMAT=I09, TITLE='Number of Households',
  DESC='Number of Households', $
  FIELD=AVGHHSZ98, ALIAS=E04, FORMAT=I09, TITLE='Average Household Size',
  DESC='Average Household Size', $
  FIELD=MEDHHI98, ALIAS=E05, FORMAT=I09, TITLE='Median Household Income',
  DESC='Median Household Income', $
  FIELD=AVGHHI98, ALIAS=E06, FORMAT=I09, TITLE='Average Household
Income',
  DESC='Average Household Income', $
  FIELD=MALEPOP98, ALIAS=E07, FORMAT=I09, TITLE='Male Population',
  DESC='Male Population', $
  FIELD=FEMPOP98, ALIAS=E08, FORMAT=I09, TITLE='Female Population',
  DESC='Female Population', $
  FIELD=P15TO1998, ALIAS=E09, FORMAT=I09, TITLE='15 to 19',
  DESC='Population 15 to 19 years old', $
  FIELD=P20TO2998, ALIAS=E10, FORMAT=I09, TITLE='20 to 29',
  DESC='Population 20 to 29 years old', $
  FIELD=P30TO4998, ALIAS=E11, FORMAT=I09, TITLE='30 to 49',
  DESC='Population 30 to 49 years old', $
  FIELD=P50TO6498, ALIAS=E12, FORMAT=I09, TITLE='50 to 64',
  DESC='Population 50 to 64 years old', $
  FIELD=P65OVR98, ALIAS=E13, FORMAT=I09, TITLE='65 and over',
  DESC='Population 65 and over', $

```

GGDEMOG Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE GGDEMOG ON 05/15/03 AT 12.26.05

```
GGDEMOG
01      S1
*****
*ST          **I
*HH          **
*AVGHHSZ98   **
*MEDHHI98    **
*            **
*****
*****
```

GGORDER Master File

```
FILENAME=GGORDER, SUFFIX=FOC,$
SEGNAME=ORDER01, SEGTYPE=S1,$
  FIELD=ORDER_NUMBER, ALIAS=ORDNO1,   FORMAT=I6,  TITLE='Order,Number',
  DESC='Order Identification Number', $
  FIELD=ORDER_DATE,   ALIAS=DATE,     FORMAT=MDY, TITLE='Order,Date',
  DESC='Date order was placed', $
  FIELD=STORE_CODE,  ALIAS=STCD,      FORMAT=A5,  TITLE='Store,Code',
  DESC='Store Identification Code (for order)', $
  FIELD=PRODUCT_CODE, ALIAS=PCD,     FORMAT=A4,  TITLE='Product,Code',
  DESC='Product Identification Code (for order)', $
  FIELD=QUANTITY,    ALIAS=ORDUNITS,  FORMAT=I8,  TITLE='Ordered,Units',
  DESC='Quantity Ordered', $
SEGNAME=ORDER02, SEGTYPE=KU, PARENT=ORDER01, CRFILE=GGPRODS, CRKEY=PCD,
CRSEG=PRODS01 , $
```

GGORDER Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE GGORDER ON 05/15/03 AT 16.45.48

```

GGORDER
01      S1
*****
*ORDER_NUMBER**
*ORDER_DATE   **
*STORE_CODE   **
*PRODUCT_CODE**
*              **
*****
          I
          I
          I
          I ORDER02
02      I KU
.....
:PRODUCT_ID   :K
:PRODUCT_DESC:
:VENDOR_CODE  :
:VENDOR_NAME  :
:              :
:.....:
    
```

GGPRODS Master File

```

FILENAME=GGPRODS, SUFFIX=FOC
SEGNAME=PRODS01, SEGTYPE=S1
FIELD=PRODUCT_ID, ALIAS=PCD, FORMAT=A4, INDEX=I, TITLE='Product,Code',
DESC='Product Identification Code', $
FIELD=PRODUCT_DESCRIPTION, ALIAS=PRODUCT, FORMAT=A16, TITLE='Product',
DESC='Product Name', $
FIELD=VENDOR_CODE, ALIAS=VCD, FORMAT=A4, INDEX=I, TITLE='Vendor ID',
DESC='Vendor Identification Code', $
FIELD=VENDOR_NAME, ALIAS=VENDOR, FORMAT=A23, TITLE='Vendor Name',
DESC='Vendor Name', $
FIELD=PACKAGE_TYPE, ALIAS=PACK, FORMAT=A7, TITLE='Package',
DESC='Packaging Style', $
FIELD=SIZE, ALIAS=SZ, FORMAT=I2, TITLE='Size',
DESC='Package Size', $
FIELD=UNIT_PRICE, ALIAS=UNITPR, FORMAT=D7.2, TITLE='Unit,Price',
DESC='Price for one unit', $
    
```

GGPRODS Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE GGPRODS ON 05/15/03 AT 12.26.05

```
GGPRODS
01      S1
*****
*PRODUCT_ID  **I
*PRODUCT_DESC**I
*VENDOR_CODE **
*VENDOR_NAME **
*           **
*****
*****
```

GGSALES Master File

```
FILENAME=GGSALES, SUFFIX=FOC
SEGNAME=SALES01, SEGTYPE=S1
FIELD=SEQ_NO, ALIAS=SEQ, FORMAT=I5, TITLE='Sequence#',
DESC='Sequence number in database', $
FIELD=CATEGORY, ALIAS=E02, FORMAT=A11, INDEX=I, TITLE='Category',
DESC='Product category', $
FIELD=PCD, ALIAS=E03, FORMAT=A04, INDEX=I, TITLE='Product ID',
DESC='Product Identification code (for sale)', $
FIELD=PRODUCT, ALIAS=E04, FORMAT=A16, TITLE='Product',
DESC='Product name', $
FIELD=REGION, ALIAS=E05, FORMAT=A11, INDEX=I, TITLE='Region',
DESC='Region code', $
FIELD=ST, ALIAS=E06, FORMAT=A02, INDEX=I, TITLE='State',
DESC='State', $
FIELD=CITY, ALIAS=E07, FORMAT=A20, TITLE='City',
DESC='City', $
FIELD=STCD, ALIAS=E08, FORMAT=A05, INDEX=I, TITLE='Store ID',
DESC='Store identification code (for sale)', $
FIELD=DATE, ALIAS=E09, FORMAT=I8YYMD, TITLE='Date',
DESC='Date of sales report', $
FIELD=UNITS, ALIAS=E10, FORMAT=I08, TITLE='Unit Sales',
DESC='Number of units sold', $
FIELD=DOLLARS, ALIAS=E11, FORMAT=I08, TITLE='Dollar Sales',
DESC='Total dollar amount of reported sales', $
FIELD=BUDUNITS, ALIAS=E12, FORMAT=I08, TITLE='Budget Units',
DESC='Number of units budgeted', $
FIELD=BUDDOLLARS, ALIAS=E13, FORMAT=I08, TITLE='Budget Dollars',
DESC='Total sales quota in dollars', $
```


GGSALES Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE GGSALES ON 05/15/03 AT 12.26.05

```

          GGSALES
01      S1
*****
*SEQ_NO   **
*CATEGORY **I
*PCD      **I
*PRODUCT  **I
*         **
*****
*****
    
```

GGSTORES Master File

```

FILENAME=GGSTORES, SUFFIX=FOC
SEGNAME=STORES01, SEGTYPE=S1
  FIELD=STORE_CODE, ALIAS=E02, FORMAT=A05, INDEX=I, TITLE='Store ID',
  DESC='Franchisee ID Code', $
  FIELD=STORE_NAME, ALIAS=E03, FORMAT=A23, TITLE='Store Name',
  DESC='Store Name', $
  FIELD=ADDRESS1, ALIAS=E04, FORMAT=A19, TITLE='Contact',
  DESC='Franchisee Owner', $
  FIELD=ADDRESS2, ALIAS=E05, FORMAT=A31, TITLE='Address',
  DESC='Street Address', $
  FIELD=CITY, ALIAS=E06, FORMAT=A22, TITLE='City',
  DESC='City', $
  FIELD=STATE, ALIAS=E07, FORMAT=A02, INDEX=I, TITLE='State',
  DESC='State', $
  FIELD=ZIP, ALIAS=E08, FORMAT=A06, TITLE='Zip Code',
  DESC='Postal Code', $
    
```

GGSTORES Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE GGSTORES ON 05/15/03 AT 12.26.05

```

          GGSTORES
01      S1
*****
*STORE_CODE **I
*STORE_NAME **
*ADDRESS1   **
*ADDRESS2   **
*         **
*****
*****
    
```

Century Corp Data Sources

Century Corp is a consumer electronics manufacturer that distributes products through retailers around the world. Century Corp has thousands of employees in plants, warehouses, and offices worldwide. Their mission is to provide quality products and services to their customers.

Century Corp is a group of data sources that contain financial, human resources, inventory, and order information. The last three data sources are designed to be used with chart of accounts data.

- CENTCOMP contains location information for stores. It consists of one segment, COMPINFO.
- CENTFIN contains financial information. It consists of one segment, ROOT_SEG.
- CENTHR contains human resources information. It consists of one segment, EMPSEG.
- CENTINV contains inventory information. It consists of one segment, INVINFO.
- CENTORD contains order information. It consists of four segments, OINFO, STOSEG, PINFO, and INVSEG.
- CENTQA contains problem information. It consists of three segments, PROD_SEG, INVSEG, and PROB_SEG.
- CENTGL contains a chart of accounts hierarchy. The field GL_ACCOUNT_PARENT is the parent field in the hierarchy. The field GL_ACCOUNT is the hierarchy field. The field GL_ACCOUNT_CAPTION can be used as the descriptive caption for the hierarchy field.
- CENTSYSF contains detail-level financial data. CENTSYSF uses a different account line system (SYS_ACCOUNT), which can be joined to the SYS_ACCOUNT field in CENTGL. Data uses "natural" signs (expenses are positive, revenue negative).
- CENTSTMT contains detail-level financial data and a cross-reference to the CENTGL data source.

CENTCOMP Master File

```

FILE=CENTCOMP, SUFFIX=FOC, FDFC=19, FYRT=00
SEGNAME=COMPINFO, SEGTYPE=S1, $
  FIELD=STORE_CODE, ALIAS=SNUM, FORMAT=A6, INDEX=I,
  TITLE='Store Id#:',
  DESCRIPTION='Store Id#', $
  FIELD=STORENAME, ALIAS=SNAME, FORMAT=A20,
  WITHIN=STATE,
  TITLE='Store,Name:',
  DESCRIPTION='Store Name', $
  FIELD=STATE, ALIAS=STATE, FORMAT=A2,
  WITHIN=PLANT,
  TITLE='State:',
  DESCRIPTION=State, $
DEFINE REGION/A5=DECODE STATE ('AL' 'SOUTH' 'AK' 'WEST' 'AR' 'SOUTH'
'AZ' 'WEST' 'CA' 'WEST' 'CO' 'WEST' 'CT' 'EAST'
'DE' 'EAST' 'DC' 'EAST' 'FL' 'SOUTH' 'GA' 'SOUTH' 'HI' 'WEST'
'ID' 'WEST' 'IL' 'NORTH' 'IN' 'NORTH' 'IA' 'NORTH'
'KS' 'NORTH' 'KY' 'SOUTH' 'LA' 'SOUTH' 'ME' 'EAST' 'MD' 'EAST'
'MA' 'EAST' 'MI' 'NORTH' 'MN' 'NORTH' 'MS' 'SOUTH' 'MT' 'WEST'
'MO' 'SOUTH' 'NE' 'WEST' 'NV' 'WEST' 'NH' 'EAST' 'NJ' 'EAST'
'NM' 'WEST' 'NY' 'EAST' 'NC' 'SOUTH' 'ND' 'NORTH' 'OH' 'NORTH'
'OK' 'SOUTH' 'OR' 'WEST' 'PA' 'EAST' 'RI' 'EAST' 'SC' 'SOUTH'
'SD' 'NORTH' 'TN' 'SOUTH' 'TX' 'SOUTH' 'UT' 'WEST' 'VT' 'EAST'
'VA' 'SOUTH' 'WA' 'WEST' 'WV' 'SOUTH' 'WI' 'NORTH' 'WY' 'WEST'
'NA' 'NORTH' 'ON' 'NORTH' ELSE ' ');,
  TITLE='Region:',
  DESCRIPTION=Region, $

```

CENTCOMP Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE CENTCOMP ON 05/15/03 AT 10.20.49

```

          COMPINFO
01      S1
*****
*STORE_CODE  **I
*STORENAME   **
*STATE       **
*            **
*            **
*****
*****

```

CENTFIN Master File

```
FILE=CENTFIN, SUFFIX=FOC, FDFC=19, FYRT=00
SEGNAME=ROOT_SEG, SEGTYPE=S4, $
FIELD=YEAR, ALIAS=YEAR, FORMAT=YY,
  WITHIN='*Time Period', $
FIELD=QUARTER, ALIAS=QTR, FORMAT=Q,
  WITHIN=YEAR,
  TITLE=Quarter,
  DESCRIPTION=Quarter, $
FIELD=MONTH, ALIAS=MONTH, FORMAT=M,
  TITLE=Month,
  DESCRIPTION=Month, $
FIELD=ITEM, ALIAS=ITEM, FORMAT=A20,
  TITLE=Item,
  DESCRIPTION=Item, $
FIELD=VALUE, ALIAS=VALUE, FORMAT=D12.2,
  TITLE=Value,
  DESCRIPTION=Value, $
DEFINE ITYPE/A12=IF EDIT(ITEM,'9$$$$$$$$$$$$$$$$') EQ 'E'
  THEN 'Expense' ELSE IF EDIT(ITEM,'9$$$$$$$$$$$$$$$$') EQ 'R'
  THEN 'Revenue' ELSE 'Asset';,
  TITLE=Type,
  DESCRIPTION='Type of Financial Line Item', $
DEFINE MOTEXT/MT=MONTH;,$
```

CENTFIN Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE CENTFIN ON 05/15/03 AT 10.25.52

```
          ROOT_SEG
01        S4
*****
*YEAR          **
*QUARTER       **
*MONTH         **
*ITEM          **
*              **
*****
*****
```

CENTHR Master File

```

FILE=CENTHR, SUFFIX=FOC,
SEGNAME=EMPSEG, SEGTYPE=S1, $
FIELD=ID_NUM, ALIAS=ID#, FORMAT=I9,
  TITLE='Employee, ID#',
  DESCRIPTION='Employee Identification Number', $
FIELD=LNAME, ALIAS=LN, FORMAT=A14,
  TITLE='Last, Name',
  DESCRIPTION='Employee Last Name', $
FIELD=FNAME, ALIAS=FN, FORMAT=A12,
  TITLE='First, Name',
  DESCRIPTION='Employee First Name', $
FIELD=PLANT, ALIAS=PLT, FORMAT=A3,
  TITLE='Plant, Location',
  DESCRIPTION='Location of the manufacturing plant',
  WITHIN='*Location', $
FIELD=START_DATE, ALIAS=SDATE, FORMAT=YYMD,
  TITLE='Starting, Date',
  DESCRIPTION='Date of employment', $
FIELD=TERM_DATE, ALIAS=TERM_DATE, FORMAT=YYMD,
  TITLE='Termination, Date',
  DESCRIPTION='Termination Date', $
FIELD=STATUS, ALIAS=STATUS, FORMAT=A10,
  TITLE='Current, Status',
  DESCRIPTION='Job Status', $
FIELD=POSITION, ALIAS=JOB, FORMAT=A2,
  TITLE=Position,
  DESCRIPTION='Job Position', $
FIELD=PAYSCALE, ALIAS=PAYLEVEL, FORMAT=I2,
  TITLE='Pay, Level',
  DESCRIPTION='Pay Level',
  WITHIN='*Wages', $
DEFINE POSITION_DESC/A17=IF POSITION EQ 'BM' THEN
  'Plant Manager' ELSE
  IF POSITION EQ 'MR' THEN 'Line Worker' ELSE
  IF POSITION EQ 'TM' THEN 'Line Manager' ELSE
  'Technician';
  TITLE='Position, Description',
  DESCRIPTION='Position Description',
  WITHIN='PLANT', $
DEFINE BYEAR/YY=START_DATE;
  TITLE='Beginning, Year',
  DESCRIPTION='Beginning Year',
  WITHIN='*Starting Time Period', $
DEFINE BQUARTER/Q=START_DATE;
  TITLE='Beginning, Quarter',
  DESCRIPTION='Beginning Quarter',
  WITHIN='BYEAR',

```

```

DEFINE BMONTH/M=START_DATE;
  TITLE='Beginning,Month',
  DESCRIPTION='Beginning Month',
  WITHIN='BQUARTER', $
DEFINE EYEAR/YY=TERM_DATE;
  TITLE='Ending,Year',
  DESCRIPTION='Ending Year',
  WITHIN='*Termination Time Period', $
DEFINE EQUARTER/Q=TERM_DATE;
  TITLE='Ending,Quarter',
  DESCRIPTION='Ending Quarter',
  WITHIN='EYEAR', $
DEFINE EMONTH/M=TERM_DATE;
  TITLE='Ending,Month',
  DESCRIPTION='Ending Month',
  WITHIN='EQUARTER', $
DEFINE RESIGN_COUNT/I3=IF STATUS EQ 'RESIGNED' THEN 1
  ELSE 0;
  TITLE='Resigned,Count',
  DESCRIPTION='Resigned Count', $
DEFINE FIRE_COUNT/I3=IF STATUS EQ 'TERMINAT' THEN 1
  ELSE 0;
  TITLE='Terminated,Count',
  DESCRIPTION='Terminated Count', $
DEFINE DECLINE_COUNT/I3=IF STATUS EQ 'DECLINED' THEN 1
  ELSE 0;
  TITLE='Declined,Count',
  DESCRIPTION='Declined Count', $
DEFINE EMP_COUNT/I3=IF STATUS EQ 'EMPLOYED' THEN 1
  ELSE 0;
  TITLE='Employed,Count',
  DESCRIPTION='Employed Count', $
DEFINE PEND_COUNT/I3=IF STATUS EQ 'PENDING' THEN 1
  ELSE 0;
  TITLE='Pending,Count',
  DESCRIPTION='Pending Count', $
DEFINE REJECT_COUNT/I3=IF STATUS EQ 'REJECTED' THEN 1
  ELSE 0;
  TITLE='Rejected,Count',
  DESCRIPTION='Rejected Count', $
DEFINE FULLNAME/A28=LNAME||', '||FNAME;
  TITLE='Full Name',
  DESCRIPTION='Full Name: Last, First', WITHIN='POSITION_DESC', $
DEFINE SALARY/D12.2=IF BMONTH LT 4 THEN PAYLEVEL * 12321
  ELSE IF BMONTH GE 4 AND BMONTH LT 8 THEN PAYLEVEL * 13827
  ELSE PAYLEVEL * 14400;;
  TITLE='Salary',
  DESCRIPTION='Salary', $

```

```
DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'  
  LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'  
  ELSE 'n/a');$
```

CENTHR Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE CENTHR ON 05/15/03 AT 10.40.34

```
          EMPSEG  
01          S1  
*****  
*ID_NUM      **  
*LNAME       **  
*FNAME       **  
*PLANT       **  
*            **  
*****  
*****
```

CENTINV Master File

```
FILE=CENTINV, SUFFIX=FOC, FDFC=19, FYRT=00
SEGNAME=INVINFO, SEGTYPE=S1, $
FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4, INDEX=I,
  TITLE='Product,Number:',
  DESCRIPTION='Product Number', $
FIELD=PRODNAME, ALIAS=PNAME, FORMAT=A30,
  WITHIN=PRODCAT,
  TITLE='Product,Name:',
  DESCRIPTION='Product Name', $
FIELD=QTY_IN_STOCK, ALIAS=QIS, FORMAT=I7,
  TITLE='Quantity,In Stock:',
  DESCRIPTION='Quantity In Stock', $
FIELD=PRICE, ALIAS=RETAIL, FORMAT=D10.2,
  TITLE='Price:',
  DESCRIPTION=Price, $
FIELD=COST, ALIAS=OUR_COST, FORMAT=D10.2,
  TITLE='Our,Cost:',
  DESCRIPTION='Our Cost:', $
DEFINE PRODCAT/A22 = IF PRODNAME CONTAINS 'LCD'
  THEN 'VCRs' ELSE IF PRODNAME
  CONTAINS 'DVD' THEN 'DVD' ELSE IF PRODNAME CONTAINS 'Camcor'
  THEN 'Camcorders'
  ELSE IF PRODNAME CONTAINS 'Camera' THEN 'Cameras' ELSE IF PRODNAME
  CONTAINS 'CD' THEN 'CD Players'
  ELSE IF PRODNAME CONTAINS 'Tape' THEN 'Digital Tape Recorders'
  ELSE IF PRODNAME CONTAINS 'Combo' THEN 'Combo Players'
  ELSE 'PDA Devices'; WITHIN=PRODTYPE, TITLE='Product Category:' , $
DEFINE PRODTYPE/A19 = IF PRODNAME CONTAINS 'Digital' OR 'DVD' OR 'QX'
  THEN 'Digital' ELSE 'Analog'; WITHIN='*Product Dimension',
  TITLE='Product Type:', $
```

CENTINV Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE CENTINV ON 05/15/03 AT 10.43.35

```
          INVINFO
01      S1
*****
*PROD_NUM    **I
*PRODNAME    **
*QTY_IN_STOCK**
*PRICE       **
*            **
*****
*****
```


CENTORD Master File

```

FILE=CENTORD, SUFFIX=FOC,
SEGNAME=OINFO, SEGTYPE=S1, $
  FIELD=ORDER_NUM, ALIAS=ONUM, FORMAT=A5, INDEX=I,
  TITLE='Order,Number:',
  DESCRIPTION='Order Number', $
  FIELD=ORDER_DATE, ALIAS=ODATE, FORMAT=YYMD,
  TITLE='Date,Of Order:',
  DESCRIPTION='Date Of Order', $
  FIELD=STORE_CODE, ALIAS=SNUM, FORMAT=A6, INDEX=I,
  TITLE='Company ID#:',
  DESCRIPTION='Company ID#', $
  FIELD=PLANT, ALIAS=PLNT, FORMAT=A3, INDEX=I,
  TITLE='Manufacturing,Plant',
  DESCRIPTION='Location Of Manufacturing Plant',
  WITHIN='*Location', $
DEFINE YEAR/YY=ORDER_DATE;,
  WITHIN='*Time Period', $
DEFINE QUARTER/Q=ORDER_DATE;,
  WITHIN='YEAR', $
DEFINE MONTH/M=ORDER_DATE;,
  WITHIN='QUARTER', $
SEGNAME=PINFO, SEGTYPE=S1, PARENT=OINFO, $
  FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4, INDEX=I,
  TITLE='Product,Number#:',
  DESCRIPTION='Product Number#', $
  FIELD=QUANTITY, ALIAS=QTY, FORMAT=I8C,
  TITLE='Quantity:',
  DESCRIPTION=Quantity, $
  FIELD=LINEPRICE, ALIAS=LINETOTAL, FORMAT=D12.2MC,
  TITLE='Line,Total',
  DESCRIPTION='Line Total', $
DEFINE LINE_COGS/D12.2=QUANTITY*COST;,
  TITLE='Line,Cost Of,Goods Sold',
  DESCRIPTION='Line cost of goods sold', $
DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'
  LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'
  ELSE 'n/a');
SEGNAME=INVSEG, SEGTYPE=DKU, PARENT=PINFO, CRFILE=CENTINV,
  CRKEY=PROD_NUM, CRSEG=INVINFO, $
SEGNAME=STOSEG, SEGTYPE=DKU, PARENT=OINFO, CRFILE=CENTCOMP,
  CRKEY=STORE_CODE, CRSEG=COMPINFO, $

```

CENTORD Structure Diagram

SECTION 01

STRUCTURE OF FOCUS

FILE CENTORD ON 05/15/03 AT 10.17.52

```

      OINFO
01      S1
*****
*ORDER_NUM    **I
*STORE_CODE   **I
*PLANT        **I
*ORDER_DATE   **
*             **
*****
      I
      +-----+
      I             I
      I STOSEG     I PINFO
02      I KU       03      I S1
.....
:STORE_CODE   :K  *PROD_NUM    **I
:STORENAME    :   *QUANTITY    **
:STATE        :   *LINEPRICE   **
:             :   *             **
:             :   *             **
:.....:
JOINED  CENTCOMP *****
                I
                I
                I
                I INVSEG
                04      I KU
.....
:PROD_NUM     :K
:PRODNAME     :
:QTY_IN_STOCK:
:PRICE        :
:             :
:.....:
                JOINED  CENTINV
```

CENTQA Master File

```

FILE=CENTQA, SUFFIX=FOC, FDFC=19, FYRT=00
SEGNAME=PROD_SEG, SEGTYPE=S1, $
  FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4, INDEX=I,
  TITLE='Product,Number',
  DESCRIPTION='Product Number', $
SEGNAME=PROB_SEG, PARENT=PROD_SEG, SEGTYPE=S1, $
  FIELD=PROBNUM, ALIAS=PROBNO, FORMAT=I5,
  TITLE='Problem,Number',
  DESCRIPTION='Problem Number',
  WITHIN=PLANT,$
  FIELD=PLANT, ALIAS=PLT, FORMAT=A3, INDEX=I,
  TITLE=Plant,
  DESCRIPTION=Plant,
  WITHIN=PROBLEM_LOCATION,$
  FIELD=PROBLEM_DATE, ALIAS=PDATE, FORMAT=YYMD,
  TITLE='Date,Problem,Reported',
  DESCRIPTION='Date Problem Was Reported', $
  FIELD=PROBLEM_CATEGORY, ALIAS=PROBCAT, FORMAT=A20, $
  TITLE='Problem,Category',
  DESCRIPTION='Problem Category',
  WITHIN=*Problem,$
  FIELD=PROBLEM_LOCATION, ALIAS=PROBLOC, FORMAT=A10,
  TITLE='Location,Problem,Occurred',
  DESCRIPTION='Location Where Problem Occurred',
  WITHIN=PROBLEM_CATEGORY,$
DEFINE PROB_YEAR/YY=PROBLEM_DATE;,
  TITLE='Year,Problem,Occurred',
  DESCRIPTION='Year Problem Occurred',
  WITHIN=*Time Period,$
DEFINE PROB_QUARTER/Q=PROBLEM_DATE;
  TITLE='Quarter,Problem,Occurred',
  DESCRIPTION='Quarter Problem Occurred',
  WITHIN=PROB_YEAR,$
DEFINE PROB_MONTH/M=PROBLEM_DATE;
  TITLE='Month,Problem,Occurred',
  DESCRIPTION='Month Problem Occurred',
  WITHIN=PROB_QUARTER,$
DEFINE PROBLEM_OCCUR/I5 WITH PROBNUM=1;,
  TITLE='Problem,Occurrence'
  DESCRIPTION='# of times a problem occurs', $
DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'
  LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'
  ELSE 'n/a');$
SEGNAME=INVSEG, SEGTYPE=DKU, PARENT=PROD_SEG, CRFILE=CENTINV,
CRKEY=PROD_NUM, CRSEG=INVINFO,$

```

CENTQA Structure Diagram

```
SECTION 01
          STRUCTURE OF FOCUS      FILE CENTQA      ON 05/15/03 AT 10.46.43

          PROD_SEG
01      S1
*****
*PROD_NUM      **I
*              **
*              **
*              **
*              **
*****
          I
          +-----+
          I              I
          I INVSEG      I PROB_SEG
02      I KU          03      I S1
.....      *****
:PROD_NUM      :K      *PROBNUM      **
:PRODNAME      :      *PLANT          **I
:QTY_IN_STOCK:      *PROBLEM_DATE**
:PRICE          :      *PROBLEM_CAT>**
:              :      *              **
:.....:      *****
JOINED CENTINV *****
```

CENTGL Master File

```
FILE=CENTGL ,SUFFIX=FOC
SEGNAME=ACCOUNTS, SEGTYPE=S1
FIELDNAME=GL_ACCOUNT, ALIAS=GLACCT, FORMAT=A7,
  TITLE='Ledger,Account', FIELDTYPE=I, $
FIELDNAME=GL_ACCOUNT_PARENT, ALIAS=GLPAR, FORMAT=A7,
  TITLE=Parent,
  PROPERTY=PARENT_OF, REFERENCE=GL_ACCOUNT, $
FIELDNAME=GL_ACCOUNT_TYPE, ALIAS=GLTYPE, FORMAT=A1,
  TITLE=Type,$
FIELDNAME=GL_ROLLUP_OP, ALIAS=GLROLL, FORMAT=A1,
  TITLE=Op, $
FIELDNAME=GL_ACCOUNT_LEVEL, ALIAS=GLLEVEL, FORMAT=I3,
  TITLE=Lev, $
FIELDNAME=GL_ACCOUNT_CAPTION, ALIAS=GLCAP, FORMAT=A30,
  TITLE=Caption,
  PROPERTY=CAPTION, REFERENCE=GL_ACCOUNT, $
FIELDNAME=SYS_ACCOUNT, ALIAS=ALINE, FORMAT=A6,
  TITLE='System,Account,Line', MISSING=ON, $
```

CENTGL Structure Diagram

SECTION 01
 STRUCTURE OF FOCUS FILE CENTGL ON 05/15/03 AT 15.18.48

```

        ACCOUNTS
    01      S1
    *****
    *GL_ACCOUNT **I
    *GL_ACCOUNT_>**
    *GL_ACCOUNT_>**
    *GL_ROLLUP_OP**
    *      **
    *****
    *****
    
```

CENTSYSF Master File

```

FILE=CENTSYSF ,SUFFIX=FOC
SEGNAME=RAWDATA ,SEGTYPE=S2
  FIELDNAME = SYS_ACCOUNT , ,A6 , FIELDTYPE=I,
    TITLE='System,Account,Line', $
  FIELDNAME = PERIOD , ,YYM , FIELDTYPE=I,$
  FIELDNAME = NAT_AMOUNT , ,D10.0 , TITLE='Month,Actual', $
  FIELDNAME = NAT_BUDGET , ,D10.0 , TITLE='Month,Budget', $
  FIELDNAME = NAT_YTDAMT , ,D12.0 , TITLE='YTD,Actual', $
  FIELDNAME = NAT_YTDBUD , ,D12.0 , TITLE='YTD,Budget', $
    
```

CENTSYSF Structure Diagram

SECTION 01
 STRUCTURE OF FOCUS FILE CENTSYSF ON 05/15/03 AT 15.19.27

```

        RAWDATA
    01      S2
    *****
    *SYS_ACCOUNT **I
    *PERIOD **I
    *NAT_AMOUNT **
    *NAT_BUDGET **
    *      **
    *****
    *****
    
```

CENTSTMT Master File

```
FILE=CENTSTMT, SUFFIX=FOC
SEGNAME=ACCOUNTS, SEGTYPE=S1
  FIELD=GL_ACCOUNT, ALIAS=GLACCT, FORMAT=A7,
    TITLE='Ledger,Account', FIELDTYPE=I, $
  FIELD=GL_ACCOUNT_PARENT, ALIAS=GLPAR, FORMAT=A7,
    TITLE=Parent,
    PROPERTY=PARENT_OF, REFERENCE=GL_ACCOUNT, $
  FIELD=GL_ACCOUNT_TYPE, ALIAS=GLTYPE, FORMAT=A1,
    TITLE=Type,$
  FIELD=GL_ROLLUP_OP, ALIAS=GLROLL, FORMAT=A1,
    TITLE=Op, $
  FIELD=GL_ACCOUNT_LEVEL, ALIAS=GLLEVEL, FORMAT=I3,
    TITLE=Lev, $
  FIELD=GL_ACCOUNT_CAPTION, ALIAS=GLCAP, FORMAT=A30,
    TITLE=Caption,
    PROPERTY=CAPTION, REFERENCE=GL_ACCOUNT, $
SEGNAME=CONSOL, SEGTYPE=S1, PARENT=ACCOUNTS, $
  FIELD=PERIOD, ALIAS=MONTH, FORMAT=YM, $
  FIELD=ACTUAL_AMT, ALIAS=AA, FORMAT=D10.0, MISSING=ON,
    TITLE='Actual', $
  FIELD=BUDGET_AMT, ALIAS=BA, FORMAT=D10.0, MISSING=ON,
    TITLE='Budget', $
  FIELD=ACTUAL_YTD, ALIAS=AYTD, FORMAT=D12.0, MISSING=ON,
    TITLE='YTD,Actual', $
  FIELD=BUDGET_YTD, ALIAS=BYTD, FORMAT=D12.0, MISSING=ON,
    TITLE='YTD,Budget', $
```

CENTSTMT Structure Diagram

SECTION 01

STRUCTURE OF FOCUS FILE CENTSTMT ON 11/06/03 AT 16.11.59

ACCOUNTS

```
01      S1
*****
*GL_ACCOUNT  **I
*GL_ACCOUNT_>**
*GL_ACCOUNT_>**
*GL_ROLLUP_OP**
*           **
*****
*****
      I
      I
      I
      I CONSOL
02      I S1
*****
*PERIOD      **
*ACTUAL_AMT  **
*BUDGET_AMT  **
*ACTUAL_YTD  **
*           **
*****
*****
```

APPENDIX B

Error Messages

Topics:

- Accessing Error Files
- Displaying Messages Online

If you need to see the text or explanation for any error message, you can display it online in your FOCUS session or find it in a standard FOCUS ERRORS file. All of the FOCUS error messages are stored in eight system ERRORS files.

Accessing Error Files

For CMS, the ERRORS files are:

- FOT004 ERRORS
- FOG004 ERRORS
- FOM004 ERRORS
- FOS004 ERRORS
- FOA004 ERRORS
- FSQXLT ERRORS
- FOCSTY ERRORS
- FOB004 ERRORS

For MVS, these files are the following members in the ERRORS PDS:

- FOT004
- FOG004
- FOM004
- FOS004
- FOA004
- FSQXLT
- FOCSTY
- FOB004

Displaying Messages Online

To display a message online, issue the following query command at the FOCUS command level

? *n*

where *n* is the message number.

The message number and text will display along with a detailed explanation of the message (if one exists). For example, issuing the following command

? 210

displays the following:

(FOC210) THE DATA VALUE HAS A FORMAT ERROR:

An alphabetic character has been found where all numerical digits are required.

Index

Symbols

- \$ screen attribute (FIDEL) 10-30
- \$\$ comment delimiter 6-8
- \$* comment delimiter 6-8
- &&FOCREBUILD variable 11-44
- &ACCEPT Dialogue Manager variable 9-213
- &CHNGD Dialogue Manager variable 9-213
- &CURSOR Dialogue Manager variable 10-33
- &CURSORAT parameter 10-36
- &DELTD Dialogue Manager variable 9-213
- &DUPLS Dialogue Manager variable 9-213
- &FOCDISORG variable 11-8
- &FOCERRNUM variable 11-44
- &FORMAT Dialogue Manager variable 9-213
- &INPUT Dialogue Manager variable 9-213
- &INVALID Dialogue Manager variable 9-213
- &NOMATCH Dialogue Manager variable 9-213
- &PFKEY field 10-23
- &REJECT Dialogue Manager variable 9-213
- &TRANS Dialogue Manager variable 9-213
- subtraction operator 8-4
- * comment delimiter 6-8
- * multiplication operator 8-3
- * option in CRTFORM command 10-47, 10-96
- *\$ comment delimiter 6-8
- ** exponentiation operator 8-3 to 8-4
- + operator 8-3 to 8-4
- . after move subcommand in SCAN 12-11
- / division operator 8-3 to 8-4
- / prefix area command in FSCAN 13-16, 13-48
- / spot marker 10-12
- <0x spot marker 10-12
- <0x> spot marker 10-12
- <n spot marker 10-12
- <n> spot marker 10-12
- = 12-26, 13-19, 13-21, 13-37, 13-42, 13-44, 13-47
 - command in FSCAN 13-37, 13-47
 - logical operator in FSCAN FIND command 13-21, 13-42
 - logical operator in FSCAN LOCATE command 13-19, 13-44
 - logical operator in SCAN LOCATE command 12-26
- == prefix area command in FSCAN 13-7
- ? 12-14, 12-41, 13-37, 13-47
 - command in FSCAN 13-37, 13-47
 - subcommand in SCAN 12-14, 12-41
- ? COMBINE command 9-197, 9-214
- ? FDT command 9-214
 - determining segment name 10-47
- ? FILE command 9-214
 - data source disorganization 11-8
 - restart when system fails 9-214
 - structural integrity of data sources 11-36
- ? PFKEY query command 10-21
- ? STAT command 9-214
- \ character in character expressions 8-20 to 8-21
- \n character in character expressions 8-20
- | and || characters in character expressions 8-19

Numerics

3270 terminals 10-25, 10-75
FIDEL screen attributes 10-25, 10-75

A

A0 variables 7-12
 passing between procedures 7-12, 7-56

Absolute File Integrity 9-207 to 9-208
 protection 11-2
 using with FSCAN 13-4, 13-38
 using with SCAN 12-2

ACCEPT attribute 9-126
 FSCAN 13-27, 13-29, 13-33
 HELPMESSAGE attribute 9-131
 transaction type 9-126

Accept List dialog box 5-49

Accept system action 5-89

Accepts Combo option in Combobox dialog box 5-71

ACCESS attribute in FSCAN 13-4

Access Files 1-4

access in FSCAN 13-4

access rights 1-4

actions 9-65 to 9-67
 DELETE 9-64, 9-67
 INCLUDE (MODIFY) 9-64 to 9-65
 system 5-87
 UPDATE 9-64, 9-66

Actions menu option in Winform Painter 5-41

ACTIVATE statement 9-199, 9-201
 MOVE option 9-201
 RETAIN option 9-201
 syntax 9-199, 9-201

active fields (MODIFY) 9-199
 conditional and non-conditional fields 10-42

ADD keyword in USE command 11-31

adding data to data sources in FSCAN 13-27

adding headers to grids 5-57

adding segment instances in SCAN 12-12

adding segments to data sources 11-13

addition in date expressions 8-13

addition operator 8-3 to 8-4

AGAIN subcommand in SCAN 12-14, 12-16

aliases for fields in SCAN 12-5 to 12-6

ALL keyword 7-22
 in COMPUTE command 7-22
 in COPY command 7-27
 in DECLARE command 7-31
 in DELETE command 7-34
 in INCLUDE command 7-51
 in NEXT command 7-66
 in REPEAT command 7-90
 in REVISE command 7-98
 in UPDATE command 7-116

ALL parameter 11-37

allocating SYSPRINT to terminal 11-5

allocating work space during rebuilding 11-5

alphanumeric expressions 8-19
 backslash (\) character 8-20 to 8-21
 concatenating strings 8-19
 escape character 8-20
 evaluating 8-20

alphanumeric format (MODIFY) 9-26
 fixed-format data sources 9-26
 passing A0 variables to procedures 7-56
 sources 9-26

altering data using SCAN 12-13

alternate file views 9-86
 MODIFY 9-71, 9-86
 with FSCAN 13-2

anchor segments 2-7

AND keyword 7-56, 12-26, 13-19, 13-21
 in FSCAN FIND command 13-21, 13-42
 in FSCAN LOCATE command 13-19, 13-44
 in LOCATE subcommand 12-26
 in MAINTAIN command 7-56
 in NEXT command 7-66

AND logical operator 8-23 to 8-24

applications 2-29
 controlling environment 7-107
 event-driven processing 2-29
 running 3-11

AQUA screen attribute (FIDEL) 10-30
 setting in Screen Painter 10-93

AS keyword 7-89, 11-31
 in RECOMPILE command 7-89
 in USE command 11-31

ASGN-FLD command in Screen Painter 10-84, 10-93

ASSIGN command in Screen Painter 10-84, 10-93

assigning values to variables (Maintain) 7-6

AT keyword 7-12

AT keyword in CALL command 7-12

attributes 2-10, 9-125
 ACCEPT 9-126
 ACCESS 13-4
 DEFINE 2-10
 FIELDTYPE=I in Master File 11-22
 HELPMESSAGE 9-131
 MISSING 9-99
 SEGTYPE 9-71

AUTOCOMMIT command 2-50

automatic CRTFORMs 10-47, 10-84, 10-96

Available Cases dialog box 5-85, 5-87

B

BACK subcommand in SCAN 12-17

background color for Winforms 5-41

background effects in FIDEL 10-28

BACKGROUND_COLOR 7-125

backing up before rebuilding 11-5

backslash (\) character in character expressions 8-21

BACKWARD command in FSCAN 13-15, 13-40
 SINGLE mode 13-26

BACKWARD setting for PFnn parameter 10-22

Backward system action 5-89

BEGIN command (Maintain) 7-10
 nested BEGIN blocks 7-11

BEGIN keyword in -CRTFORM command 10-4, 10-79

BLIN screen attribute (FIDEL) 10-30

blinking fields (FIDEL) 10-25
 dynamically changing 10-30

block mode 2-28

BLUE screen attribute (FIDEL) 10-30
 setting in Screen Painter 10-93

Boolean expressions 8-23

borders for Winforms 5-9
 adding around controls 5-59
 adding in Winform Properties dialog box 4-12, 5-38
 adding to button control 4-38, 5-62
 adding to grid 5-57
 adding to radio buttons 5-74

BOX command in Screen Painter 10-84, 10-92

brackets for Winform Painter 4-4, 5-26

Index

- branching in FIDEL 10-23, 10-36
- browser control 5-58
- Browser menu option in Winform Painter 5-58
- browsers (Winforms) 3-15
- business logic 1-6
- Button command in Winform Painter 4-35
- button control 4-35, 5-7, 5-60
 - shortcuts 5-83
- Button menu option in Winform Painter 5-60

C

- C prefix area command in FSCAN 13-23, 13-41, 13-48
- CALL command 1-7, 2-19, 7-12, 7-58
 - MAINTAIN command and 7-58
 - passing variables 2-20
- CANCEL setting for PFnn parameter 10-22
- canceling commands (Maintain) 6-7
- canvas in Winform Painter 5-14
- CAR data source A-14 to A-16
- caret symbol (spot marker) 10-12
- CASE command 7-15 to 7-16
 - PERFORM command and 7-17
- case logic (MODIFY) 9-132, 9-158
 - applications 9-148
 - bad values 9-148, 9-156
 - branching 9-137, 9-144
 - cases 9-135
 - ENDCASE statement 9-133
 - GOTO statement 9-137
 - IF statement 9-137, 9-142
 - incoming values 9-148, 9-155
 - MATCH statement 9-137, 9-146
 - NEXT statement 9-148 to 9-149
 - offering user selections 9-148, 9-153

- case logic (MODIFY) (*continued*)
 - ON INVALID phrase 9-137, 9-148
 - PERFORM statement 9-137, 9-139
 - repeating groups 9-31
 - REPOSITION statement 9-148 to 9-149
 - rules 9-135, 9-137, 9-144, 9-149
 - START case 9-137
 - syntax 9-133
 - TRACE facility 9-157
 - transaction data sources 9-148, 9-154
 - unique segments 9-148, 9-151
 - validation tests 9-137, 9-148
 - with FIDEL 10-57

- case sensitivity 6-2
 - specifying in FIDEL 10-14

- cases (Maintain) 7-15
 - applying to form-level triggers 5-85
 - calling using COMPUTE 7-26
 - editing in Winform Painter 4-42, 5-78
 - opening in Winform Painter 5-79
 - return values 7-19
 - TOP 9-133
 - Top function 7-19

- Cases menu in Winform Painter 4-42, 5-78

- CDN (Continental Decimal Notation) 8-7
 - setting in a Maintain procedure 7-110

- CENTCOMP data source A-39

- center justification 4-37, 5-61, 5-74
 - for button text 4-37, 5-61
 - for radio button text 5-74

- CENTFIN data source A-38

- CENTGL data source A-49

- CENTHR data source A-38

- CENTINV data source A-38

- CENTORD data source A-38

- CENTQA data source A-38

- CENTSTMT A-50
- CENTSYSF data source A-49
- Century Corp data sources A-38 to A-41, A-43 to A-48
- CHANGE 12-13, 12-18, 13-32, 13-40
 - command in FSCAN 13-32, 13-40
 - subcommand in SCAN 12-13, 12-18
- change verification 2-44, 2-46
 - DB2 data sources 2-50
- ChangeColBcolor 7-125
- ChangeColFcolor 7-125
- changing data using SCAN 12-13
- changing screen attributes (FIDEL) 10-30
- changing Winform properties dynamically 7-125
- character expressions 8-19
 - \ character 8-20 to 8-21
 - \n character 8-20
 - | and || characters 8-19
 - backslash (\) character 8-20 to 8-21
 - concatenating strings 8-19
 - double quotation marks 8-20
 - escape character 8-20
 - evaluating 8-20
 - single quotation marks 8-20
- character set in Winform Painter 4-4, 5-26
- character strings 8-20 to 8-21
- check box control 5-5, 5-64
- CHECK FILE PICTURE command 10-47
 - determining segment name 10-47
- CHECK statement 9-207
 - checkpoint facility 9-207
 - MODIFY 9-207
- CHECK subcommand of REBUILD facility 11-33
- Checkbox menu option in Winform Painter 5-64
- checking pointers in data sources 11-33
- Checkpoint facility 9-207
 - comparing CHECK and COMMIT 9-214
 - placement in case logic requests 9-135
- CHILD command in FSCAN 13-23, 13-41
 - SINGLE mode 13-26
- child procedures 2-22
- classes 2-57
 - class editor 2-60
 - defining 2-60, 7-38
 - DESCRIBE command 2-60
 - functions 2-57
 - inheritance 2-57
 - libraries 2-62, 7-64
 - subclasses 2-57
 - superclasses 2-57
 - variables 2-57
- CLEA screen attribute (FIDEL) 10-30
- CLEAR command 9-189
 - COMBINE 9-189
- CLEAR keyword 4-46, 7-105, 10-74
 - in CRTFORM command 10-74
 - in STACK CLEAR command 4-46, 7-105
- clearing screen in FIDEL 10-74, 10-80
- client/server deployment 1-2
- CLOSE keyword in WINFORM command 7-120
- Close system action 5-89
- CLOSE_ALL keyword in WINFORM command 7-120
- Close_form system action 5-89
- CO logical operator 13-19
 - in FSCAN FIND command 13-21, 13-42
 - in FSCAN LOCATE command 13-19, 13-44
- coloring fields (FIDEL) 10-25
 - dynamically changing 10-30
 - in Screen Painter 10-93

Index

- colors in Winform Painter 5-9
 - setting for controls 5-51
 - setting for Winforms 5-41
 - setting via WINFORM SET 7-125
- columns 2-7
 - in Winform Painter grid 4-30, 5-55
 - redefining 2-20
 - temporary 2-10
 - user-defined 2-10
 - virtual 2-10
- columns in stacks with virtual fields 7-25
- COMBINE command 9-189 to 9-190, 9-197
 - ? COMBINE 9-190, 9-197
 - clearing 9-190
 - combining structures 9-190, 9-196
 - compared to JOIN 9-190, 9-197
 - PREFIX parameter 9-190, 9-194
 - syntax 9-191
 - TAG parameter 9-190 to 9-191, 9-193
 - with FSCAN 13-2
- combo box control 5-8, 5-68
- Combobox menu option in Winform Painter 5-68
- comma-delimited data sources (MODIFY) 9-22, 9-35
 - activating fields 9-199, 9-201
 - date formats 9-35
 - default 9-35
 - identifying values 9-37
 - log files 9-126
 - MATCH statement 9-40
 - missing values 9-39
 - NEXT statement 9-40
 - ON ddname option 9-35
 - PROMPT statement 9-41, 9-50
- command buttons (Winforms) 5-7
- command failure in data sources 2-37
- command types 6-7, 13-7
 - command-line 13-7
 - immediate 13-7
 - multi-line 6-7
 - non-immediate 13-7
 - prefix area 13-7, 13-48
- command-line commands in FSCAN 13-7
- commands 6-7
 - / prefix area 13-16, 13-48
 - ? 13-37, 13-47
 - ? COMBINE 9-190
 - ? FILE 9-214
 - AUTOCOMMIT 2-50
 - BACKWARD 13-15, 13-40
 - C prefix area 13-23, 13-41, 13-48
 - CALL 1-7, 2-19
 - canceling 6-7
 - CHANGE 13-32, 13-40
 - CHILD 13-23, 13-41
 - CLEAR 9-189
 - COMBINE 9-189, 9-197, 13-2
 - COMMIT 2-36, 2-44, 3-11, 9-207, 9-209, 9-214
 - COMPILE 1-7, 9-198
 - COMPUTE 2-10, 2-15
 - D prefix area 13-35, 13-41, 13-48
 - DECLARE 2-63
 - DELETE 2-32, 3-11, 13-35, 13-41
 - DESCRIBE 2-60
 - DISPLAY 13-41
 - DOWN 13-14, 13-41
 - END 3-5, 13-38, 13-41
 - EX 9-14
 - EX MSETUP 4-4
 - EXEC 1-7
 - FILE 9-207, 13-38, 13-41
 - FIND 13-21, 13-42
 - FIRST 13-18, 13-43
 - FOR prefix 2-4
 - FORWARD 13-14, 13-43
 - FS 13-5
 - GOTO END 2-23

commands (*continued*)

HELP 13-39, 13-43
 HOLD 9-163
 I prefix area 13-27, 13-43, 13-48
 IF 9-137, 9-142
 INCLUDE 2-32, 3-11
 INFER 2-7
 JOIN 9-107
 JUMP 13-25, 13-43
 K prefix area 13-33, 13-45, 13-48
 KEY 13-33, 13-45
 LAST 13-18, 13-43
 last 13-37
 LEFT 13-15, 13-43
 LOCATE 13-19, 13-44
 MAINTAIN 2-20, 3-5
 MATCH 2-7, 2-30, 3-6
 MNTCON 4-48
 MODIFY 9-13
 MODULE 2-62
 MOVE 9-201
 MPAINT 4-20
 MULTIPLE 13-26, 13-45
 NEXT 2-7, 2-30, 3-6
 PARENT 13-25, 13-45
 PERFORM 4-42
 previous 13-37
 QQUIT 13-4, 13-38, 13-45
 QUIT 13-38, 13-45
 R prefix area 13-30, 13-46, 13-48
 REDEFINES 9-94
 REPEAT 2-14
 repeating last 13-37
 REPLACE 13-31, 13-46
 REPLACE KEY 13-35, 13-46
 REPOSITION 2-22, 3-6 to 3-7, 4-46
 RESET 13-30, 13-33, 13-46
 REVISE 2-32, 3-11
 RIGHT 13-2, 13-47
 ROLLBACK 2-36, 3-11
 RUN 1-7, 9-198
 SAVE 13-37, 13-47

commands (*continued*)

SCAN FILE 12-4
 SINGLE 13-26, 13-47
 STACK CLEAR 4-46
 STACK SORT 2-14
 TOP 13-18, 13-47
 truncating 13-7
 UPDATE 2-32, 3-11, 4-42
 USE 2-44, 2-48, 13-2
 WINFORM 3-8
 WRITE 2-33

commands in Winform Painter 4-27

Button 4-35
 Copy 4-33
 Edit Object 4-27
 Field 4-20, 4-22
 Grid 4-29
 Move 4-27
 Resize 4-27
 Save 4-19
 Save As 4-19
 Terminal 4-4
 Text 4-33

comment delimiters 6-8

dollar sign asterisk 6-8
 double dollar sign 6-8

comments 6-8

generated by Winform Painter 5-29
 Maintain procedures 6-6, 6-8

COMMIT command (Maintain) 2-36, 2-44, 3-11

DB2 data sources 2-50
 defining a logical transaction 2-34
 setting parameter from a Maintain procedure
 7-110

COMMIT command (MODIFY) 7-20, 9-207, 9-209, 9-214

Absolute File Integrity 9-214
 compared to CHECK 9-214
 MATCH statement 9-214 to 9-215
 NEXT statement 9-214 to 9-215
 system failure 9-214 to 9-215

Index

- COMPILE command (Maintain) 1-7, 7-21, 7-61
- COMPILE command (MODIFY) 9-198
 - RUN 9-198
- compiled calculations (MODIFY) 9-93
- COMPUTE command (Maintain) 2-10, 2-15, 4-44, 4-46, 7-22
 - creating user-defined fields 7-25
 - defining stack columns 2-10
 - IF command and 7-50
- COMPUTE statement (MODIFY) 9-93
 - changing incoming data 9-98
 - compilation 9-93
 - deactivating 9-206
 - FIND function 9-107 to 9-108
 - LOOKUP function 9-107
 - MATCH statement 9-93, 9-97
 - multiple statements 9-93, 9-96
 - NEXT statement 9-93, 9-97
 - non-data source fields 9-93, 9-98
 - placement 9-93, 9-96
- concatenating index data sources 11-30
- concurrent transactions 2-42
 - change verification 2-45
 - FOCUS Database Server 2-45
 - processing 2-44 to 2-45, 2-50
- conditional actions (Maintain) 7-8
- conditional expressions 8-25
 - COMPUTE commands 7-50
 - IF statement 9-137, 9-142
 - VALIDATE statement 9-99, 9-102
- conditional fields (MODIFY) 9-33, 10-14, 10-42
 - adding segment instances 9-33
 - fixed-format transaction data sources 9-33
 - text 9-26
- CONTAINS logical operator 8-24, 12-26
 - in FSCAN FIND command 13-21, 13-42
 - in FSCAN LOCATE command 13-19, 13-44
 - in NEXT WHERE phrase 7-72
- Continental Decimal Notation (CDN) 8-7
- CONTINUE TO method 9-71 to 9-72
 - NEXT statement 9-88, 9-91
- control box (Winform Painter) 5-5
- controls 4-20, 4-27
 - copying 4-33, 5-32
 - creating 5-44
 - deleting 5-32
 - editing 4-27, 5-31
 - grouping 5-59
 - moving 4-27, 5-31
 - resizing 4-27, 5-32
 - setting color in Winform Painter 5-51
- converting legacy dates to smart dates 11-41
- COPY command 7-27
- Copy command in Winform Painter 4-33
- Copy menu option in Winform Painter 5-32
- Copy To menu option in Winform Painter 5-35
- copying cases/functions in Winform Painter 5-81
- copying controls in Winform Painter 4-33, 5-32
- copying forms in Winform Painter 5-35
- copying lines in Screen Painter 10-89
- copying stack rows 7-26
- COURSES data source A-20
- CREATE command 11-2
- creating controls 5-44
- creating data sources 11-2
- creating Maintain procedures in Winform Painter 5-19
- creating objects (Maintain) 7-5
- creating stacks with INFER 7-54
- creating variables (Maintain) 7-5

- creating Winforms 5-34
- cross-referenced data sources 2-7
- CRTFORM command 10-5, 10-7 to 10-8
 - BEGIN and END keywords 10-4, 10-79
 - TYPE keyword 10-81
- CRTFORM command 10-5, 10-7 to 10-8
 - * option 10-47, 10-96
 - CLEAR/NOCLEAR keywords 10-74
 - LINE keyword 10-51
 - TYPE keyword 10-77
 - WIDTH and HEIGHT keywords 10-75
 - with FIXFORM command 10-45
- CRTFORM statement 9-19
 - active fields 9-199 to 9-200
 - HELPMESSAGE attribute 9-131
 - log files 9-126
 - messages 9-131
 - multiple record processing 9-159, 9-165
 - REPEAT statement 9-159, 9-169
- CRTFORM subcommand in SCAN 12-20
- CRTFORMs 10-1, 10-5
 - clearing screen 10-80
 - cursor position 10-33
 - defining fields 10-9, 10-14, 10-28, 10-88
 - defining menus 10-23, 10-36
 - defining variables 10-78
 - differences in MODIFY and Dialogue Manager 10-11
 - filling out 10-6
 - invoking 10-8
 - looping 10-79
 - PF keys 10-20
 - resizing message area 10-81
 - setting screen attributes 10-25, 10-30
 - spot markers 10-12
 - using Screen Painter 10-84
- CRTFORMs 10-1 to 10-2, 10-7
 - clearing screen 10-74
 - cursor position 10-33
 - defining fields 10-9, 10-14, 10-28, 10-42, 10-59, 10-88
 - defining menus 10-23, 10-36
 - determining beginning line 10-51
 - differences in MODIFY and Dialogue Manager 10-11
 - filling out 10-6
 - generating automatically 10-47, 10-96
 - handling errors 10-68
 - invoking 10-8
 - PF keys 10-20
 - resizing message area 10-77
 - setting screen attributes 10-25, 10-30
 - sizing 10-75
 - spot markers 10-12
 - using Screen Painter 10-84
 - with MODIFY case logic 10-57
- CurGrdColNum 7-125
- CurGrdRowNum 7-125
- Current Area 2-16
- current instance in FSCAN 13-7, 13-16
 - viewing in SINGLE mode 13-26
- CURRENT keyword in COPY command 7-27
- current position in data source 12-4
- cursor position 9-132, 9-167, 10-33
 - CURSORINDEX variable 9-159, 9-167
 - FIDEL facility 10-33
- CURSOR variable (MODIFY) 9-159, 9-167, 10-33
- CURSORAT field (MODIFY) 10-36
- CURSORINDEX variable (MODIFY) 10-33, 10-65
- CurStkColNum 7-125
- CurStkRowNum 7-125
- Custom character set dialog box in Winform Painter 4-4, 5-26
- customizing Winform Painter 4-4, 5-25

D

D prefix area command in FSCAN 13-35, 13-41, 13-48

D. prefix for display fields in FIDEL 10-15
dynamically changing to T. 10-30
setting in Screen Painter 10-93

data 9-1
COMBINE command 9-190, 9-197
entry fields 9-200
from multiple sources 9-190, 9-197

data continuity 2-23

data entry fields (FIDEL) 10-14
designating in Screen Painter 10-93

data fields 12-6, 12-10
deleting in SCAN 12-13
displaying names 12-10
identifying in SCAN 12-6
in FSCAN 13-3

DATA keyword 7-22
in COMPUTE command 7-22
in DECLARE command 7-31

DATA keyword (MODIFY) 9-13

data source operations 3-11

data source stacks 6-3, 7-54
copying rows 7-26
libraries and 7-64
naming 6-3
virtual fields 7-25

data source types 1-4
3-level 9-71, 9-81
disorganized 11-8
externally indexed 11-30
fixed-format 9-24, 9-29, 9-34
FOCUS 9-189, 13-2
free-format 9-35
HLIPRINT 9-9
joined 2-30
multi-path 9-71
multiple in one request 9-190, 9-197

data sources 1-4, 3-11, A-1
adding data 13-27
change verification 2-44, 2-46
child procedures and 2-22
concatenating 11-30
creating 11-2
editing with FSCAN 13-2
evaluating success 2-32
indexing 11-13, 11-22, 11-26
integrity 9-8, 11-8, 11-33
libraries and 7-64
migrating to Fusion 11-47
navigating 7-75, 12-6
operations 2-3, 3-11
optimizing 11-8
partitioning 11-5
position 2-22, 2-31, 2-36, 3-6 to 3-7, 7-97
processing transactions 2-34, 3-11, 9-34
reading 1-4, 2-7, 2-30, 9-19
rebuilding 11-5, 11-8
security 11-5
sharing 2-44 to 2-45
source segment 11-30
structure 11-30, 11-33, 13-10
target segment 11-30
time stamp 11-39
transactions 2-34
updating 7-98, 7-116, 13-29
verifying 11-33
writing 1-5, 2-7, 2-32

DATA statement 9-56
case logic requests 9-135
ON ddname option 9-56
VIA program option 9-56

data types 2-57
built-in 2-57
classes 2-57
matching in function parameters 7-18
synonyms 7-39

database integrity in FSCAN 13-4

- Database Server 9-8
- database stacks 2-2
 - creating 2-6
 - defining 2-7
- databases 11-1
- date and time expressions 8-8
- date expressions 8-8
 - addition and subtraction in 8-13
 - components 8-11
 - constants in 8-10
 - evaluating 8-9
 - extracting 8-11
 - formats 8-8, 8-10
 - manipulating in date format 8-10
 - operand format 8-12
- DATE NEW subcommand in REBUILD facility 11-41
- date stamp for data sources 11-39
- DATEDISPLAY parameter 7-110
- dates 9-29
 - base date 9-29
 - comma-delimited data sources 9-36
 - describing (MODIFY) 9-29
 - internal format 9-29
 - MODIFY transactions 9-26
 - natural literal 9-29
- date-time data types 8-14, 8-16
 - assigning 8-16
 - comparing 8-16
 - describing 8-14
 - functions 8-17
 - ISO standards 8-18
 - missing values and 8-16
- date-time formats 8-14
- date-time values 8-14
 - assigning 8-14
- DB2 data sources 2-50
 - concurrent transactions 2-50
 - reading 1-4
 - transaction processing 2-50
 - writing 1-5
- DB2 row locks 2-50, 2-55
- DBA (database administration) 1-4 to 1-5
 - COMBINE command 9-190
 - FSCAN facility 13-4
- DBMS_ERRORCODE command 7-107
- DEACTIVATE statement 9-206
 - COMPUTES option 9-206
 - INVALID option 9-206
 - RETAIN option 9-206
 - syntax 9-206
- deactivating fields 9-206
- DECLARE command 2-63, 7-31
- DECODE function (MODIFY) 9-99, 9-106
- Default Button option for button control 4-38, 5-62
- default settings for PF keys (FIDEL) 10-21
- default value for entry fields 5-47
- DEFCENT parameter 7-110
 - with REBUILD NEW DATE 11-41
- DEFINE attribute 2-10
 - in Master File 7-25
- DEFINEd fields in FSCAN 13-3
- DELETE action (MODIFY) 9-64, 9-67
 - descendant segments 9-71, 9-76
 - NEXT statement 9-148 to 9-149
 - segment instances 9-67
- DELETE command 2-32, 3-11, 7-33 to 7-34
 - deleting data 7-33
 - in FSCAN 13-35, 13-41
- DELETE command in Screen Painter 10-89

Index

- Delete menu option in Winform Painter 5-32, 5-36
- DELETE subcommand in SCAN 12-21
- deleting columns from grid 5-57
- deleting fields and segments in data sources 11-13
- deleting fields and segments in SCAN 12-13
- deleting functions/cases in Winform Painter 5-82
- deleting segments in data sources 11-8
- deleting Winforms 5-36
- descendant segments 9-71, 9-76
 - 3-level data sources 9-71, 9-81
 - displaying in FSCAN 13-23
 - matching across segments 9-71, 9-76
 - MODIFY 9-71, 9-76
 - multi-path data sources 9-71, 9-82
 - updating 9-71, 9-76
- DESCRIBE command 7-38
- destination stack 4-12, 4-23, 4-31, 5-39, 5-47, 5-58
- DFC keyword 7-22
 - in COMPUTE command 7-22
 - in DECLARE command 7-31
- DFC parameter 7-110
 - with REBUILD NEW DATE 11-41
- dialog boxes (Winform Painter) 5-2
 - Accept List 5-49
 - Custom Character set 5-26
 - Edit Column 5-56
 - PFKeys 5-63
 - PFKeys help 5-83
 - Set Colors 5-51
 - System setup 5-26
- Dialogue Manager 10-4
 - difference in FIDEL with MODIFY 10-11
 - using with FIDEL 10-4, 10-78
- Dialogue Manager variables 9-213
 - &ACCEPT 9-213
 - &CHNGD 9-213
 - &DELTD 9-213
 - &DUPLS 9-213
 - &FORMAT 9-213
 - &INPUT 9-213
 - &INVALID 9-213
 - &NOMATCH 9-213
 - &REJECT 9-213
 - &TRANS 9-213
- directory paths 8-20
- disorganized data sources 11-8
 - finding 11-33
 - fixing 11-8
- DISPLAY 12-10, 12-22, 13-41
 - command in FSCAN 13-41
 - subcommand in SCAN 12-10, 12-22
- display fields (FIDEL) 10-14
 - changing from turnaround fields 10-30
 - conditional and non-conditional 10-43
 - designating in Screen Painter 10-93
- DIV operator 8-3 to 8-4
- division operator 8-3 to 8-4
- dollar sign asterisk comment delimiter 6-8
- double dollar sign comment delimiter 6-8
- double-precision (MODIFY) 9-26
 - fixed-format data sources 9-26
- DOWN command in FSCAN 13-14, 13-41
- drop boxes (Winform Painter) 5-9
- DROP keyword 7-12, 7-42
 - in CALL command 7-12
- DROP TABLE command from Maintain 7-108
- DUMP option in REORG subcommand 11-14
- DUPLICATE command in Screen Painter 10-89

duplicate field names (MODIFY) 9-22
 duplicate field values in repeating groups 10-70
 duplicate key fields with FSCAN 13-3
 duplicate names 6-3
 dynamically changing Winform properties 7-125

E

ECHO keyword (MODIFY) 9-13, 9-209
 Edit Column dialog box 4-32, 5-56
 Edit menu in Winform Painter 4-27, 4-33, 5-30 to 5-32
 Edit Object command in Winform Painter 4-27
 Edit Object menu option in Winform Painter 5-31
 EDIT operator 8-21
 editing data sources with FSCAN 13-2
 Editor menu option in Winform Painter 5-29
 EDUCFILE data source A-9 to A-10
 ELSE keyword 7-47 to 7-48, 8-25
 in conditional expressions 8-25
 in IF command 7-47
 embedded data (MODIFY) 9-117, 9-120
 EMGSRV parameter 7-110
 EMPDATA data source A-21 to A-22
 EMPLOYEE data source A-3, A-5 to A-6
 END command 3-5, 7-40
 in FSCAN 13-38, 13-41
 in Screen Painter 10-84, 10-97
 END keyword 9-4
 ending a prompting session 9-41, 9-46
 in -CRTFORM command 10-4, 10-79
 in FSCAN 9-4, 13-5
 in GOTO command 7-44
 in MODIFY 9-4, 9-13
 in SCAN 9-4
 position in request 9-14

END setting for PFnn parameter 10-22
 END subcommand in SCAN 12-14, 12-23
 ENDBEGIN keyword 7-10
 ENDCASE 7-15 to 7-16, 7-44
 command 7-15 to 7-16
 keyword in GOTO command 7-44
 ENDCASE statement (MODIFY) 9-133
 CASE statement 9-133
 GOTO statement 9-137
 HELPMESSAGE attribute 9-131
 IF statement 9-137, 9-142
 logging 9-126
 PERFORM statement 9-137, 9-139
 user-specified 9-117, 9-125
 ENDDESCRIBE keyword in DESCRIBE command 7-38
 END-OF-CHAIN message in SCAN 12-25
 ENDREPEAT 7-44, 7-90
 command 7-90
 keyword in GOTO command 7-44
 ENDREPEAT statement (MODIFY) 9-159
 GOTO command 9-159, 9-161
 REPEAT statement 9-159, 9-161
 ENGINE command 7-108
 entering FSCAN 13-5
 entry field control 4-20, 4-22, 5-3, 5-45
 entry fields (FIDEL) 10-14
 designating in Screen Painter 10-93
 EQ logical operator 8-24, 12-26
 in FSCAN FIND command 13-21, 13-42
 in FSCAN LOCATE command 13-19, 13-44
 in NEXT WHERE phrase 7-72
 EQ_MASK logical operator in NEXT WHERE phrase 7-72

Index

- error messages 7-107 to 7-108
 - retrieving from DBMSs 7-107 to 7-108
 - error messages (MODIFY) 9-129
 - controlling display 9-129
 - errors in FIDEL 10-68
 - escape characters 8-20 to 8-21
 - in character expressions 8-20
 - evaluating success of data source commands 2-32
 - event handlers 5-83
 - event-driven development 1-2, 1-5, 2-25, 5-1
 - event-driven processing 1-2, 2-25, 2-29
 - EX command 4-48, 7-41, 7-62
 - executing MODIFY requests 9-14
 - EX MSETUP command 4-4, 5-26
 - EXCEEDS logical operator 7-72
 - in NEXT WHERE phrase 7-72
 - EXEC command 1-7, 7-41
 - executing compiled procedures (Maintain) 7-102
 - executing uncompiled procedures (Maintain) 7-41
 - EXIT keyword in GOTO command 7-44
 - EXIT statement (MODIFY) 9-135
 - GOTO statement 9-138
 - Exit system action 5-89
 - exiting FSCAN 13-38
 - exiting SCAN 12-14
 - exiting Screen Painter 10-97
 - exiting Winform Painter 4-19, 5-12
 - EXITREPEAT keyword in GOTO command 7-44
 - EXITREPEAT statement (MODIFY) 9-159, 9-161
 - explicit Top function 7-19
 - exponentiation operator 8-3 to 8-4
 - expressions 8-1 to 8-2
 - Boolean 8-23
 - character 8-19
 - conditional 7-50, 8-25
 - date and time 8-8
 - default values 8-26
 - limits in Maintain 8-2 to 8-3
 - logical 8-22
 - null values 8-26
 - numeric 8-3
 - relational 8-22
 - extended attributes (FIDEL) 10-25
 - EXTERNAL INDEX subcommand of REBUILD facility 11-26
 - extracting substrings 8-20
 - EXTTERM parameter 10-25
 - screen attributes 9-125
-
- F**
- facilities 9-11, 9-189
 - Checkpoint 9-207
 - ECHO 9-209
 - FIDEL 9-51
 - FSCAN 13-1 to 13-2
 - MODIFY 9-1 to 9-2
 - SCAN 12-1
 - Simultaneous Usage (SU) 9-8, 9-135
 - TRACE 9-13, 9-157
 - FALSE value for logical expressions 8-22 to 8-23
 - FDT query command 10-47
 - determining segment name 10-47
 - FETCH SQL command equivalent (Maintain) 7-65
 - FIDEL (FOCUS Interactive Data Entry Language) 10-1 to 10-2
 - clearing screen 10-74, 10-80
 - controlling PF keys 10-20
 - cursor position 10-33
 - defining fields 10-9, 10-14, 10-28, 10-42, 10-59

- FIDEL (FOCUS Interactive Data Entry Language)
 - (continued)
 - defining menus 10-23, 10-36
 - determining beginning line 10-51
 - generating forms automatically 10-47, 10-96
 - handling errors 10-68
 - invoking 10-5
 - positioning text and fields 10-12
 - resizing message area 10-77, 10-81
 - Screen Painter 10-84
 - sizing screens 10-75
 - specifying screen attributes 10-25, 10-30
 - spot markers 10-12
 - using screens 10-6
 - using with Dialogue Manager 10-4, 10-11, 10-78
 - using with MODIFY 10-2, 10-11, 10-41, 10-57
- FIDEL command in Screen Painter 10-84, 10-95
- Field command in Winform Painter 4-20, 4-22
- field formats (MODIFY) 9-26
 - COMPUTE statement 9-93, 9-96
 - fixed-format data sources 9-26
 - VALIDATE statement 9-99
- Field menu option in Winform Painter 5-45
- FieldLeft system action 5-89
- FieldRight system action 5-89
- fields (Dialogue Manager) 10-9
 - defining in -CRTFORM 10-9
 - defining in Screen Painter 10-88
 - setting length 10-93
- fields (Maintain) 2-10
 - displaying to end users 5-46, 5-65, 5-67, 5-71, 5-74 to 5-75, 5-77
 - generating list in comments 5-29
 - naming 6-3
 - null values 7-124, 8-26
 - passing 2-20
 - temporary 2-10
 - virtual 2-10
- fields (MODIFY) 9-4
 - conditional 9-26
 - defining in CRTFORM 10-9
 - defining in Screen Painter 10-88
 - displaying multiple fields in FIDEL 10-59
 - HOLD COUNT 9-171
 - HOLD INDEX 9-171
 - SCREEN INDEX 9-171
 - setting length 10-93
 - using labeled fields (FIDEL) 10-28
- FIELDTYPE=I attribute in Master File 11-22
- FILE command in FSCAN 13-38, 13-41
- FILE keyword 11-2, 12-4
 - in CREATE command 11-2
 - in MAINTAIN command 7-56
 - in MODIFY command 9-13
 - in SCAN FILE command 12-4
- File menu in Winform Painter 4-19 to 4-20, 5-18 to 5-22, 5-24 to 5-25, 5-30
- FILE subcommand in SCAN 12-14, 12-23
- FILES keyword in MAINTAIN command 7-56
- FINANCE data source A-18
- FIND command 13-21
 - in FSCAN 13-21, 13-42
- FIND function 9-99
 - MODIFY 9-99, 9-103, 9-107 to 9-109
 - NOT FIND function 9-107 to 9-108
 - validating data 9-109
- FIRST command in FSCAN 13-18, 13-43
- fixed-format data sources (MODIFY) 9-24
 - activating fields 9-199, 9-201
 - conditional fields 9-26
 - log files 9-126
 - moving through a record 9-25
 - syntax 9-22
 - transaction field formats 9-26
 - X-n notation 9-24 to 9-25

Index

- FIXFORM command with FIDEL 10-45
- FIXFORM statement 9-20
 - from HOLD (MODIFY) 9-22
- FLAS screen attribute (FIDEL) 10-30
- flashing fields (FIDEL) 10-25
 - dynamically changing 10-30
- floating-point fields 9-26
 - MODIFY fixed-format data sources 9-26
- flow of control (Maintain) 1-2, 2-18, 7-12
 - branching 2-18
 - CALL command 7-12
 - CASE command 7-15
 - forms 7-123
 - GOTO command 7-44
 - IF command 7-47
 - looping 2-18, 7-90
 - nesting 2-18
 - ON MATCH command 7-82
 - ON NEXT command 7-83
 - ON NOMATCH command 7-84
 - ON NONEXT command 7-85
 - PERFORM command (Maintain) 7-87
 - REPEAT command 7-90
 - triggering 2-18
 - WINFORM command 7-123
- FOCCOMP ddname/file type 7-61
- FocCount variable 2-12, 3-15, 7-42
- FocCurrent variable 7-42
 - with COMMIT command 7-20
- FOCDISORG amper variable 11-8
- FOCERRNUM amper variable 11-44
- FocError variable 7-42
- FocErrorRow variable 2-33, 7-43
- FOCEXEC business logic 1-6
- FOCEXEC file extension 5-10
- FocFetch variable 7-43
- FocIndex variable 2-12, 3-15, 7-43
 - setting with a list box 5-66
 - setting with combo box 5-68
- FOCREBUILD amper variable 11-44
- FOCSET command 7-110
- FOCURRENT variable 9-9
- FOCUS data sources 9-189
 - combining (MODIFY) 9-189 to 9-190, 9-196
 - editing with FSCAN 13-2
- FOCUS Database Server 2-44, 2-48, 9-8
 - with FSCAN 13-2
- FOCUS Screen Painter 10-84
- FOR command prefix 2-4
- FOR keyword 7-27
 - in COPY command 7-27
 - in DELETE command 7-34
 - in INCLUDE command 7-51
 - in NEXT command 7-66, 7-71
 - in REVISE command 7-98
 - in UPDATE command 7-116
- BACKGROUND_COLOR 7-125
- formats 9-26
 - handling entry errors in FIDEL 10-68
 - MODIFY fixed-format data sources 9-26
 - MODIFY temporary fields 9-93
- form-level triggers 5-83
- forms (-CRTFORMS) 10-1, 10-5
 - allocating space for variables 10-78
 - clearing screen 10-80
 - cursor position 10-33
 - defining fields 10-9, 10-14, 10-28, 10-88
 - defining menus 10-23, 10-36
 - differences in MODIFY and Dialogue Manager 10-11
 - filling out 10-6

- forms (-CRTFORMs) (*continued*)
 - invoking 10-8
 - looping 10-79
 - PF keys 10-20
 - resizing message area 10-81
 - setting screen attributes 10-25, 10-30
 - spot markers 10-12
 - using Screen Painter 10-84
- forms (CRTFORMs) 10-1 to 10-2, 10-7
 - clearing screen 10-74
 - cursor position 10-33
 - defining fields 10-9, 10-14, 10-28, 10-42, 10-59, 10-88
 - defining menus 10-23, 10-36
 - determining beginning line 10-51
 - differences in MODIFY and Dialogue Manager 10-11
 - filling out 10-6
 - generating automatically 10-47, 10-96
 - handling errors 10-68
 - invoking 10-8
 - PF keys 10-20
 - resizing message area 10-77
 - setting screen attributes 10-25, 10-30
 - sizing 10-75
 - spot markers 10-12
 - using Screen Painter 10-84
 - with MODIFY case logic 10-57
- forms (Maintain) 7-125
 - changing properties dynamically 7-125
 - displaying at run time 7-120
 - displaying default values 7-124
 - dynamically changing properties 7-125
 - flow of control 7-123
 - libraries and 7-64
- forms in Maintain (Winforms) 5-33
 - copying 5-35
 - creating 5-34
 - deleting 5-36
 - importing 5-22
 - moving 5-43
- forms in Maintain (Winforms) (*continued*)
 - properties 5-37
 - renaming 5-36
 - resizing 5-42
 - setting background color 5-41
 - specifying system actions 5-87
 - specifying triggers 5-85
 - switching between 5-34
 - title 5-38
 - validating user entries 5-49
- Forms menu in Winform Painter 5-33 to 5-37, 5-41 to 5-43
- FORWARD command in FSCAN 13-14, 13-43
 - SINGLE mode 13-26
- FORWARD setting for PFnn parameter 10-22
- Forward system action 5-89
- frame control 5-59
- Frame menu option in Winform Painter 5-59
- FREEFORM statement 9-35
- free-format data sources (MODIFY) 9-35
 - activating fields 9-199, 9-201
 - date formats 9-35
 - default 9-35
 - FREEFORM statement 9-37
 - identifying fields 9-37
 - identifying values 9-37
 - MATCH statement 9-40
 - missing values 9-39
 - NEXT statement 9-40
 - ON ddname option 9-35
 - PROMPT statement 9-41, 9-50
- frequency of messages during REBUILD 11-6
- FROM keyword 7-12
 - in CALL command 7-12
 - in COPY command 7-27
 - in DELETE command 7-34
 - in INCLUDE command 7-51
 - in MAINTAIN command 7-56

FROM keyword (*continued*)
 in MATCH command 7-59
 in REVISE command 7-98
 in UPDATE command 7-116

FS command 13-5

FSCAN facility 13-1 to 13-2, 13-4
 defining current instance 13-16
 displaying descendant segments 13-23
 entering 13-5
 exiting 13-38
 FOCUS structures 13-10
 getting help 13-39
 PF keys 13-48
 restrictions 13-2
 saving your work 13-37
 scrolling 13-14
 security 13-4
 show lists 13-5
 syntax summary 13-40
 using the screen 13-7

function keys FSCAN 13-48

function keys in Winform Painter 4-15, 5-14, 5-17, 5-83
 assigning to triggers 5-83
 generating list in comments 5-29

functions (Maintain) 7-2, 8-17
 calling using COMPUTE 7-26
 calling using PERFORM 7-87 to 7-88
 date-time 8-17
 defining 7-15
 editing in Winform Painter 4-42, 5-78
 naming 6-3
 passing parameters 7-18
 return values 7-19
 Top 7-19

functions (MODIFY) 9-99
 DECODE 9-99, 9-103, 9-106
 FIND 9-86, 9-99, 9-103, 9-107 to 9-108
 LOOKUP 9-107, 9-110

Fusion data sources 11-47
 migrating to FOCUS 11-47
 multi-dimensional indexes 11-47

G

GE logical operator 8-24, 12-26
 in FSCAN FIND command 13-21, 13-42
 in FSCAN LOCATE command 13-19, 13-44

Gen Master menu option in Winform Painter 5-77

Gen Segment menu option in Winform Painter 5-75

Generated Code Section in Maintain procedure 4-42

GET keyword in WINFORM command 7-120

GET_PREMATCH command 7-112

GETHOLD statement 9-171, 9-178

GGDEMOG data source A-33

GGORDER data source A-33

GGPRODS data source A-33

GGSALES data source A-33

GGSTORES data source A-33

global variables 7-33

Gotham Grinds data sources A-33 to A-37

GOTO command (Maintain) 7-44
 data source commands and 7-45
 ENDCASE command and 7-46
 PERFORM command and 7-46, 7-88

GOTO END command (Maintain) 2-23

GOTO statement (MODIFY) 9-137 to 9-138
 ENDREPEAT statement 9-159, 9-161
 EXIT 9-88 to 9-89, 9-137 to 9-138
 EXITREPEAT statement 9-159, 9-161
 MATCH and NEXT statements 9-137, 9-146
 ON INVALID phrase 9-99, 9-104
 rules governing branching 9-137, 9-144
 syntax 9-138

Graphical User Interface (GUI) 1-2

GRAY screen attribute (FIDEL) 10-93
 setting in Screen Painter 10-93

GREE screen attribute (FIDEL) 10-30

Grid command in Winform Painter 4-29

grid control 4-29, 5-54

Grid menu option in Winform Painter 5-54

grids in Winforms 3-18

grouping controls with frames 5-59

groups of fields 10-59
 handling errors 10-70
 with case logic 10-64

GT logical operator 8-24, 12-26
 in FSCAN FIND command 13-21, 13-42
 in FSCAN LOCATE command 13-19, 13-44

GUI (Graphical User Interface) 1-2

H

HEIGHT keyword in CRTFORM command 10-75

Height option in Combobox dialog box 5-71

HELP 9-132
 command in FSCAN 13-39, 13-43
 command in Screen Painter 10-84
 key (MODIFY) 9-131 to 9-132
 setting for PFnn parameter 10-22

Help menu 5-83

HELPMESSAGE attribute 9-131
 CRTFORMs 9-131
 setting PF key 10-22

HIDE keyword in WINFORM command 7-120

HIGH screen attribute (FIDEL) 10-30
 setting in Screen Painter 10-93

HIGHEST keyword in STACK SORT command 7-106

highlighting fields (FIDEL) 10-25
 changing in Screen Painter 10-93
 dynamically changing 10-30

HLIPRINT data source 9-9

HOLD command 9-163
 MODIFY 9-159, 9-161, 9-163
 with FIDEL 10-61

HOLDCOUNT variable 9-159, 9-164, 9-168

I

I prefix area command in FSCAN 13-27, 13-43, 13-48
 defining current instance 13-16

IBM 3270 terminals 10-5, 10-25
 FIDEL screen attributes 10-25, 10-75

IF command (Maintain) 4-44, 4-46, 7-47
 keyword in conditional expressions 8-25

IF command (MODIFY) 9-137, 9-142
 MATCH and NEXT 9-137, 9-146
 rules governing branching 9-137, 9-144

immediate commands in FSCAN 13-7

implicit Top function 7-19

IMPORT keyword in MODULE command 7-64

Import menu option in Winform Painter 5-22

importing modules 2-62, 7-64
 restrictions 7-64

IN logical operator in NEXT WHERE phrase 7-72

inactive fields (MODIFY) 9-199
 conditional and non-conditional fields 10-42

INCLUDE action (MODIFY) 9-64
 descendant segments 9-71, 9-76
 NEXT statement 9-88 to 9-89
 type S0 segments 9-71, 9-83
 WITH-UNIQUES method 9-71, 9-74

Index

- INCLUDE command 2-32, 3-11, 7-50 to 7-51
 - adding data 7-50
 - data source position 7-53
 - null values 7-53
 - unique segments 7-51
 - INDEX keyword in USE command 11-31
 - INDEX subcommand of REBUILD facility 11-22
 - indexed fields 9-107
 - FIND function 9-107 to 9-108
 - indexing data sources 11-13, 11-22, 11-26
 - using DFSORT sort library 11-22
 - using SYNCSORT sort library 11-22
 - using VMSORT sort library 11-22
 - INFER command 2-7, 7-54 to 7-55
 - inheritance 2-57
 - initialization 9-171
 - Scratch Pad Area 9-171
 - input area in FSCAN 13-7
 - INPUT subcommand in SCAN 12-24
 - INSERT command in Screen Painter 10-89
 - INTE screen attribute (FIDEL) 10-30
 - integer fields 9-26
 - fixed-format data sources 9-26
 - MODIFY fixed-format data sources 9-29
 - integrity of data sources 9-8, 11-8, 11-33
 - integrity of database in FSCAN 13-4
 - intensifying fields (FIDEL) 10-25
 - dynamically changing 10-30
 - internal date format 9-29
 - INTO keyword 7-12
 - in CALL command 7-12
 - in COPY command 7-27
 - in INFER command 7-54
 - in MAINTAIN command 7-56
 - in MATCH command 7-59
 - in NEXT command 7-66
 - INVALID keyword in ON INVALID GOTO 10-69
 - with FIDEL 10-69
 - INVALID transaction type 9-207
 - INVE screen attribute (FIDEL) 10-30
 - setting in Screen Painter 10-93
 - inverting fields (FIDEL) 10-25
 - dynamically changing 10-30
 - setting in Screen Painter 10-93
 - IS logical operator in NEXT WHERE phrase 7-72
 - IS_NOT logical operator in NEXT WHERE phrase 7-72
 - issuing DROP TABLE command 7-108
 - ITEMS data source A-30
- ## J
-
- JOBFILE data source A-7 to A-8
 - JOIN command 9-107
 - COMBINE command 9-190, 9-197
 - LOOKUP function 9-107, 9-110
 - with MODIFY 9-6
 - joins 2-7
 - JUMP 13-23, 13-25
 - command in FSCAN 13-25, 13-43
 - subcommand in SCAN 12-9, 12-25
 - justification for button text 4-37, 5-61
 - justification for radio button text 5-74
- ## K
-
- K prefix area command in FSCAN 13-33, 13-45, 13-48
 - defining current instance 13-16
 - KEEP keyword 7-44
 - in CALL command 7-12
 - in GOTO command 7-44
 - KEEP option 2-23

KEY 13-33 to 13-35, 13-45 to 13-46
 command in FSCAN 13-33, 13-45
 keyword in FSCAN REPLACE KEY command
 13-35, 13-46
 keyword in REPLACE command 12-31

key fields 2-7
 changing in FSCAN 13-2 to 13-3, 13-33
 editing with SCAN 12-2, 12-31
 protecting in Winforms 4-24, 4-32, 5-48, 5-56
 requirements 2-7

key fields (MODIFY) 9-71
 segments with multiple keys 9-71, 9-85
 segments with no keys 9-71, 9-83
 updating 9-59

keys 9-131 to 9-132
 HELP 9-131 to 9-132
 PF (FSCAN) 9-132

KEYS keyword in CRTFORM *KEYS command 10-47

keywords 4-42, 4-46, 9-13, 10-4, 12-26, 13-5
 AND 12-26
 AND in FSCAN FIND command 13-21, 13-42
 AND in FSCAN LOCATE command 13-19, 13-44
 BEGIN 10-4, 10-79
 CLEAR 4-46
 DATA 9-13
 ECHO 9-13, 9-209
 END 9-3
 FILE 9-13, 12-4
 HEIGHT 10-75
 INVALID 10-69
 KEY 13-35, 13-46
 KEY in REPLACE command 12-31
 KEYS 10-47
 LINE 10-51
 LOWER 10-14
 NOCLEAR 10-74
 NONKEYS 10-47
 ON 9-13
 OR in FSCAN FIND command 13-21, 13-42
 OR in FSCAN LOCATE command 13-21, 13-44

keywords (*continued*)
 RETAIN 9-201
 SEG 10-47
 SEG in FSCAN command 13-5
 SHOW 4-42
 SHOW option in FSCAN command 13-5
 TYPE in -CRTFORM command 10-81
 TYPE in CRTFORM command 10-77
 UPPER 10-14
 VIA 9-13, 9-56
 WHERE 4-46
 WIDTH 10-75

L

labeled fields (FIDEL) 10-28
 changing in Screen Painter 10-93
 dynamically changing 10-30

labels for case logic (MODIFY) 9-132, 9-137

LANGUAGE parameter 7-110

Language Wizard 7-2

LAST command in FSCAN 13-18, 13-37, 13-43

last subcommand in SCAN 12-14

LE logical operator 8-24, 12-26
 in FSCAN FIND command 13-21, 13-42
 in FSCAN LOCATE command 13-19, 13-44

LEDGER data source A-17

LEFT command in FSCAN 13-15, 13-43

left justification 4-37, 5-61, 5-74
 for button text 4-37, 5-61
 for radio button text 5-74

Left system action 5-89

legacy date fields 11-41
 converting 11-41

length of columns in grid control 4-32, 5-56

length of fields 4-23, 5-48
 setting in Screen Painter (FIDEL) 10-93

Index

- length of variables in -CRTFORMs 10-78
- libraries (Maintain) 2-62
 - class definitions 2-62
 - importing 2-62, 7-64
 - nesting 2-62
 - restrictions 7-64
- limits (MODIFY) 9-190
 - combined structures 9-190, 9-197
 - fixed-formats 9-22
 - length of TYPE lines 9-117 to 9-118
 - messages displayed in a case 9-135
 - number of cases 9-135
 - PROMPT text 9-41, 9-45
- line feeds in character expressions 8-20
- LINE keyword in CRTFORM command 10-51
- list box control 5-4, 5-66
- Listbox menu option in Winform Painter 5-66
- LOAD option in REORG subcommand 11-14
- load procedures A-1
- local variables 7-33
- LOCATE 12-8, 12-26, 13-19, 13-44
 - command in FSCAN 13-19, 13-44
 - subcommand in SCAN 12-8, 12-26
- LOG command with FIDEL 10-73
- log files (Maintain) 7-103
 - SAY command 7-103
 - segment and stack values 7-104
 - TYPE command 7-113
- LOG statement 9-126
- logging FOCUS Database Server actions 9-9
- logging transactions (MODIFY) 9-126
 - controlling rejection messages 9-129
 - CRTFORM 9-126
 - FIDEL 10-73
 - FIXFORM statement 9-126
 - logging transactions (MODIFY) (*continued*)
 - FREEFORM statement 9-126
 - NOMATCH phrase 9-126
 - placing in case logic requests 9-135
 - PROMPT 9-126
 - record length of log file 9-126
- logical expressions 8-22
 - Boolean expressions 8-23
 - evaluating 8-23
 - operators 8-23 to 8-24
 - relational expressions 8-22
- logical operators 8-23
- logical transactions 2-34, 2-36, 3-11
 - broadcast commit 2-36, 2-40
 - concurrent 2-42
 - data source position 2-36, 2-38
 - evaluating success 2-36, 2-41
 - FocCurrent variable 2-41
 - multiple data source types 2-36, 2-40
 - rolling back 2-36
 - size 2-36
 - spanning procedures 2-36
- LOOKUP function 9-107, 9-110
 - data source values used for searching 9-113
 - deactivated fields 9-199 to 9-200
 - next highest or lowest value 9-107, 9-115
 - segment types accessible 9-107, 9-110
 - VALIDATE statement 9-107, 9-116
- looping in a -CRTFORM 10-79
- looping in Maintain language 7-5
- loops 7-92
 - branching out of 7-96
 - ending 7-96
 - simple 7-92
- LOWER keyword in [-]CRTFORM command 10-14
- lowercase in FIDEL 10-14
- LT logical operator 8-24, 12-26
 - in FSCAN FIND command 13-21, 13-42
 - in FSCAN LOCATE command 13-19, 13-44

- M**
-
- Maintain 1-2
 - performance 1-7
 - restricting access to data 1-4
 - MAINTAIN command 3-5, 7-56 to 7-57
 - calling a procedure from another procedure 7-58
 - passing variables 2-20
 - specifying data sources 7-57
 - MAINTAIN file extension 5-10
 - Maintain functions 7-2
 - calling using COMPUTE 7-26
 - calling using PERFORM 7-87 to 7-88
 - defining 7-15
 - passing parameters 7-18
 - return values 7-19
 - Top 7-19
 - Maintain language 6-1, 7-2
 - case sensitivity 6-2
 - class libraries 7-9
 - commands 6-7, 7-1 to 7-2, 7-5
 - comments 6-6, 6-8
 - conditional actions 7-8
 - displaying forms 7-5
 - encrypting files 7-5
 - function libraries 7-9
 - loops 7-5
 - manipulating stacks 7-6
 - messages and logs 7-9
 - multi-line commands 6-7
 - naming rules 6-2 to 6-3, 6-5
 - reading data 7-7
 - transferring control 7-3
 - variables 7-5 to 7-6
 - writing transactions 7-8
 - Maintain procedure file 5-10
 - Maintain procedures 6-6, 6-8
 - blank lines 6-6
 - calling 2-19
 - comments 6-8
 - compiling 7-61, 7-89
 - passing data 2-20
 - running 4-48, 7-4, 7-102
 - MARK subcommand in SCAN 12-27
 - markers (MODIFY) 9-120
 - Master Files 1-4, A-1
 - adding fields to Winforms 5-77
 - in Winform Painter 4-9, 5-10, 5-24
 - MATCH command 2-7, 2-30, 7-59 to 7-61
 - collecting values 3-6
 - defining stack columns 2-7
 - NEXT command and 7-75
 - REPOSITION command and 7-97
 - MATCH keyword in ON MATCH command 7-82
 - MATCH statement (MODIFY) 9-58, 9-217
 - 3-level FOCUS structure 9-71, 9-81
 - activating fields 9-68, 9-70, 9-199, 9-206
 - adding segment instances 9-64 to 9-65, 9-71, 9-76
 - alternate file views 9-71, 9-86
 - case logic 9-132
 - COMMIT subcommand 9-207, 9-209
 - COMPUTE statement 9-92
 - CONTINUE 9-71, 9-76
 - CONTINUE TO method 9-71 to 9-72
 - deactivating fields 9-68, 9-70, 9-206
 - defaults 9-59, 9-63
 - deleting segment instances 9-64, 9-67
 - FIXFORM statement 9-34
 - FREEFORM statement 9-35
 - GOTO, PERFORM, and IF statements 9-137, 9-146
 - INCLUDE action 9-64
 - MATCH/ON NOMATCH statement 9-59, 9-61, 9-71, 9-76
 - multi-path data sources 9-71, 9-82

Index

- MATCH statement (MODIFY) (*continued*)
 - NEXT statement 9-88, 9-91
 - PROMPT statement 9-41, 9-49
 - REPEAT statement 9-159, 9-161
 - TYPE statement 9-117
 - updating key fields 9-59
 - updating segment instances 9-64, 9-66
 - VALIDATE statement 9-99
 - WITH-UNIQUES method 9-71, 9-74, 9-88
- MDINDEX subcommand of REBUILD facility 11-47
- member functions 2-57
 - defining with DESCRIBE 7-38
 - inheritance 2-57
- member variables 2-57
 - defining with DESCRIBE 7-38
 - inheritance 2-57
- memory management 2-23
- menu options in Winform Painter
 - Actions 5-41
 - Browser 5-58
 - checkbox 5-64
 - Combobox 5-68
 - Copy To 5-35
 - Delete 5-32, 5-36
 - Editor 5-31
 - Frame 5-59
 - Gen Master 5-77
 - Gen Segment 5-75
 - Import 5-22
 - Listbox 5-66
 - New (Maintain procedure) 5-19
 - New (Winform) 5-34
 - OPEN 5-20
 - Pictorial View 5-29
 - Preferences 5-25
 - Properties 5-48
 - Radio Group 5-72
 - Regen 5-24
 - Rename 5-36
 - Select Master 5-24
 - Size 5-42
 - Switch To 5-34
 - Triggers 5-41
 - Zoom 5-42
- menu options in Winform Painter (*continued*)
 - Size 5-42
 - Switch To 5-34
 - Triggers 5-41
 - Zoom 5-42
- menus in FIDEL 10-23, 10-36
- menus in Winform Painter 4-15, 5-14
- MESSAGE parameter 7-110
- messages frequency during REBUILD 11-6
- methods 9-71
 - CONTINUE TO 9-71 to 9-72
 - WITH-UNIQUES 9-71, 9-74
- MIGRATE subcommand of REBUILD facility 11-47
- MISSING attribute 7-124, 8-26
 - date-time data type and 8-16
 - VALIDATE 9-99, 9-106
- MISSING constant 8-26 to 8-27
- missing data (MODIFY) 9-33
 - comma-delimited data 9-39
 - fixed-format data sources 9-26
 - prompted data 9-41
 - validation tests 9-99, 9-105
- missing data and INCLUDE command 7-53
- missing data in SCAN 12-11
- MISSING keyword 7-22
 - in COMPUTE command (Maintain) 7-22
 - in DECLARE command (Maintain) 7-31
- MNTCON COMPILE command 7-61
- MNTCON EX command 4-48, 7-62
- MNTCON RUN command 7-63
- MOD operator 8-4 to 8-5

MODIFY 9-1

- ? COMBINE command 9-190, 9-197
- Absolute File Integrity 9-207 to 9-208, 9-214
- activating fields 9-199, 9-201
- advanced facilities 9-189
- cancelling transactions 9-41, 9-46
- case logic 9-132
- checkpoint 9-207
- COMBINE command 9-190, 9-197
- COMMIT subcommand 9-214
- compiled requests 9-198
- COMPUTE statement 9-93
- correcting field values 9-41, 9-47
- cross-referenced segments 9-107, 9-110
- CRTFORM statement 9-51
- DATA statement 9-56, 9-135
- deactivating fields 9-206
- DECODE function 9-99, 9-106
- defining incoming data 9-19
- descendant segments 9-71, 9-76
- describing date fields 9-29
- difference in FIDEL with Dialogue Manager 10-11
- displaying messages 9-117, 9-131
- ECHO facility 9-209
- entering no data 9-41, 9-48
- executing requests 9-14
- FIND function 9-99, 9-103, 9-107
- FIXFORM from HOLD 9-22
- HELPMESSAGE attribute 9-131
- logging transactions 9-126
- LOOKUP function 9-107, 9-110
- managing transactions 9-214
- MATCH statement 9-59
- multiple data sources in one request 9-190, 9-197
- multiple record processing 9-158
- NEXT statement 9-88
- positioning text 9-122
- procedure execution 9-209, 9-213
- prompting 9-41, 9-43, 9-45, 9-199, 9-201, 9-206
- query commands 9-214

MODIFY (*continued*)

- repeating a previous response 9-41, 9-48
- request syntax 9-217
- ROLLBACK subcommand 9-214
- Scratch Pad Area 9-158, 9-171, 9-187
- SET FIELDNAME command 9-190, 9-192
- SET TEXTFIELD command 9-52, 9-54
- SORTHOLD statement 9-171, 9-187
- sorting the Scratch Pad Area 9-187
- START statement 9-57
- statistical variables 9-213
- TAG parameter 9-190, 9-193
- TED (text editor) 9-52
- text fields 9-52, 9-55
- TRACE facility 9-157
- transaction fields in combined data sources 9-190, 9-193
- TYPE statement 9-117
- unique segments 9-68, 9-71, 9-88, 9-91
- using with FIDEL 10-2, 10-41
- VALIDATE statement 9-99
- WITH-UNIQUES method 9-71, 9-74, 9-88, 9-92

MODIFY applications 2-49

- sharing data sources 2-49

MODIFY command 9-13

- executing online 9-16

MODIFY facility 9-1 to 9-2

modifying segments 9-68, 9-71, 9-76, 9-82 to 9-83, 9-85

modular processing 1-2, 7-12

MODULE command 2-62

MODULE IMPORT command 7-64

modules 2-62, 7-64

- importing 2-62, 7-64

MORE=> symbol in FSCAN 13-7

MOVE command (MODIFY) 9-201

Move command in Winform Painter 4-27

Index

- Move menu option in Winform Painter 5-31, 5-43
 - MOVE subcommand in SCAN 12-28
 - MOVIES data source A-29
 - moving segment instances in SCAN 12-13
 - MPAINT command 4-20, 5-11
 - multi-path data sources 9-71
 - MODIFY 9-71, 9-82
 - MULTIPLE command in FSCAN 13-26, 13-45
 - multiple record processing (MODIFY) 9-158
 - CURSOR variable 9-159, 9-167
 - CURSORINDEX variable 9-159, 9-167
 - GETHOLD statement 9-171, 9-178
 - HOLD phrase 9-159, 9-163
 - HOLDCOUNT variable 9-159, 9-164, 9-171
 - HOLDINDEX field 9-171
 - initialization 9-171
 - manual methods 9-171 to 9-172, 9-180
 - REPEAT statement 9-158 to 9-159, 9-161
 - REPEATCOUNT variable 9-159, 9-164
 - REPOSITION statement 9-171
 - Scratch Pad Area 9-158 to 9-159, 9-165
 - SCREENINDEX field 9-171, 9-176
 - segments 9-171, 9-181, 9-183
 - validating data 9-159, 9-167
 - multiple record processing phases 9-159 to 9-160, 9-165, 9-168, 9-171 to 9-172, 9-176, 9-178
 - multiple-key segments 9-71, 9-85
 - multiplication operator 8-3 to 8-4
 - multiply occurring fields 10-59
 - handling errors 10-70
 - with case logic 10-64
- ## N
-
- names 6-3
 - naming conventions 6-5
 - reserved words 6-5
 - native-mode arithmetic 8-6
 - navigating in data sources in SCAN 12-6
 - NE logical operator 8-24, 12-26
 - in FSCAN FIND command 13-21, 13-42
 - in FSCAN LOCATE command 13-19, 13-44
 - in NEXT WHERE phrase 7-72
 - NE_MASK logical operator in NEXT WHERE phrase 7-72
 - NEEDS keyword 7-22, 7-31
 - in COMPUTE command 7-22
 - in DECLARE command 7-31
 - nested BEGIN blocks 7-10 to 7-11
 - nesting IF commands 7-48
 - New (Maintain procedure) menu option in Winform Painter 5-19
 - New (Winform) menu option in Winform Painter 5-34
 - new data sources 11-2
 - new segments in SCAN 12-12
 - NEXT 12-8, 12-29, 13-14, 13-45
 - command in FSCAN 13-14, 13-45
 - subcommand in SCAN 12-8, 12-29
 - NEXT command (Maintain) 2-7, 2-30, 4-42, 7-65 to 7-67
 - collecting values 3-8
 - copying data 7-70
 - data source navigation 7-75 to 7-77, 7-79
 - defining stack columns 2-7
 - loading multi-path transaction data 7-71
 - reading data 7-75
 - REPOSITION command 7-97
 - retrieving rows 7-74
 - set-based processing 2-30, 3-6
 - unique segments 7-82
 - with MATCH command 7-75
 - with multiple rows 7-71
 - with path instances 7-77

- NEXT keyword in ON NEXT command 7-83
 - NEXT statement (MODIFY) 9-88, 9-148 to 9-149
 - actions 9-68, 9-70
 - ACTIVATE and DEACTIVATE statements 9-199 to 9-200
 - adding segment instances 9-89, 9-148 to 9-149
 - case logic 9-148 to 9-149
 - CONTINUE TO method 9-88, 9-91
 - GOTO and IF statements 9-137, 9-146
 - GOTO EXIT statement 9-88 to 9-89
 - GOTO, PERFORM, and IF statements 9-137, 9-146
 - LOOKUP function 9-107, 9-110
 - looping 9-148 to 9-149
 - non-case logic requests 9-89 to 9-90
 - PROMPT statement 9-41, 9-49
 - REPEAT statement 9-159, 9-161
 - REPOSITION statement 9-148 to 9-149
 - restrictions 9-88 to 9-89, 9-148 to 9-149
 - syntax 9-89
 - TYPE statement 9-120
 - unique segments 9-88, 9-91
 - VALIDATE statement 9-99, 9-105
 - NOCLEAR keyword in CRTFORM command 10-74
 - handling errors 10-69
 - NODATA parameter 7-124
 - null representation 8-26
 - setting in a Maintain procedure 7-110
 - NODI screen attribute (FIDEL) 10-30
 - setting in Screen Painter 10-93
 - nodisplay fields (FIDEL) 10-25
 - dynamically changing 10-30
 - setting in Screen Painter 10-93
 - NOMATCH keyword in ON NOMATCH command 7-84
 - NOMATCH transaction type (MODIFY) 9-126
 - rejection message 9-129
 - non-conditional fields (MODIFY) 9-26, 10-14, 10-42
 - NONEXT keyword in ON NONEXT command 7-86
 - non-intermediate commands in FSCAN 13-7
 - non-key segments 9-71, 9-83
 - NONKEYS keyword in CRTFORM * NONKEYS command 10-47
 - NOT FIND function 9-107 to 9-108
 - NOT logical operator 8-23 to 8-24
 - NOT_IN logical operator in NEXT WHERE phrase 7-72
 - null values 7-22, 7-53, 7-124, 8-26
 - COMPUTE command 7-22
 - expressions 8-26
 - forms 7-124
 - INCLUDE command 7-53
 - testing 8-27
 - numeric expressions 8-3
 - Continental Decimal Notation (CDN) 8-7
 - DIV 8-3 to 8-4
 - evaluating 8-6
 - identical operand formats 8-6
 - MOD 8-4
 - operand formats 8-6 to 8-7
 - operators 8-3 to 8-4
 - truncating decimal values 8-7
- O**
-
- object-oriented development 1-2
 - objects 2-57
 - declaring 2-63, 7-31
 - defining 2-57
 - developing 2-57
 - orientation 2-57
 - variable editor 2-63
 - objects (controls) 4-27, 4-33, 5-31 to 5-32
 - copying 4-33, 5-32
 - creating 5-44
 - deleting 5-32

objects (controls) (*continued*)

- editing 4-27, 5-31
- grouping 5-59
- moving 4-27, 5-31
- resizing 4-27, 5-32
- setting color in Winform Painter 5-51

Objects menu in Winform Painter 5-44

- Browser 5-58
- Button 4-35, 5-60
- Checkbox 5-64
- Combobox 5-68
- Field 4-20, 4-22, 5-45
- Frame 5-59
- Gen Master 5-77
- Gen Segment 5-75
- Grid 4-29, 5-54
- Listbox 5-66
- Radio Group 5-72
- Text 4-33, 5-53

OFF keyword 7-22, 7-31

- in COMPUTE command 7-22
- in DECLARE command 7-31

OM logical operator 13-19, 13-21, 13-42, 13-44

- in FSCAN FIND command 13-21, 13-42
- in FSCAN LOCATE command 13-19, 13-44

OMITS logical operator 8-24, 12-26

- in FSCAN FIND command 13-21, 13-42
- in FSCAN LOCATE command 13-19, 13-44
- in NEXT WHERE phrase 7-72

ON INVALID GOTO with FIDEL 10-69

ON INVALID phrase 9-99, 9-104

- case logic 9-137, 9-148
- TYPE 9-117 to 9-118

ON keyword 7-22, 7-31, 7-56, 7-113, 9-13

- in COMPUTE command 7-22
- in DECLARE command 7-31
- in MAINTAIN command 7-56
- in MODIFY 9-13
- in TYPE command 7-113

ON MATCH command 7-82 to 7-83

ON MATCH phrase (MODIFY) 9-59

- actions 9-59 to 9-60, 9-64, 9-68 to 9-69
- COMMIT subcommand 9-214
- CONTINUE 9-71, 9-76
- CONTINUE TO method 9-71 to 9-72
- defaults 9-59, 9-63, 9-71, 9-76
- ROLLBACK subcommand 9-214 to 9-215
- TED (text editor) 9-52

ON MATCH/NOMATCH phrase (MODIFY) 9-61

ON NEXT command 7-83 to 7-84

ON NEXT phrase (MODIFY) 9-88 to 9-89

- adding segment instances 9-88 to 9-89, 9-148 to 9-149
- COMMIT subcommand 9-214 to 9-215
- CONTINUE TO method 9-88, 9-91
- ROLLBACK subcommand 9-214 to 9-215
- TED (text editor) 9-52

ON NOMATCH command 7-84 to 7-85

ON NOMATCH phrase (MODIFY)

- COMMIT subcommand 9-215
- defaults 9-59, 9-63, 9-71, 9-76
- ROLLBACK subcommand 9-214 to 9-215
- TED (text editor) 9-52

ON NONEXT command 7-85 to 7-86

ON NONEXT phrase (MODIFY) 9-88 to 9-89

- illegal actions 9-88 to 9-89
- ROLLBACK subcommand 9-214 to 9-215
- TED (text editor) 9-52

ON TABLE SET command 7-105

OnTop system action 5-89

Open menu option in Winform Painter 5-20

opening FSCAN 13-5

operand format for dates 8-12

optimizing data sources 11-8

- options 2-23
 - KEEP 2-23
 - RESET 2-23
 - OR keyword 13-19, 13-21, 13-42, 13-44
 - in FSCAN FIND command 13-21, 13-42
 - in FSCAN LOCATE command 13-19, 13-44
 - OR logical operator 8-23 to 8-24
- P**
-
- packed-decimal fields (MODIFY) 9-26
 - fixed-format data sources 9-26
 - PAINT command (TED) 10-84
 - parameters 7-18
 - &CURSORAT 10-36
 - EXTTERM 9-125
 - PFnn 10-22, 10-87
 - PREFIX 9-190
 - SET 10-33, 10-36
 - SET 10-22, 10-25, 10-87, 12-2, 12-33
 - SET EXTTERM 9-117, 9-125
 - SET FIELDNAME 9-190, 9-192
 - SET TEXTFIELD 9-52, 9-54
 - SHADOW 12-2, 12-33
 - PARENT command in FSCAN 13-25, 13-45
 - SINGLE mode 13-26
 - partitioning data sources 11-5
 - PASS setting 7-110 to 7-111
 - Paste button in Grid dialog box 5-57
 - PATHCHECK environment variable 2-47
 - PERFORM command (Maintain) 4-42, 7-17, 7-87
 - data source commands and 7-88
 - GOTO command and 7-46, 7-88
 - nesting 7-88
 - PERFORM statement (MODIFY) 9-137, 9-139
 - MATCH and NEXT statements 9-137, 9-146
 - period (.) after move subcommand in SCAN 12-11
 - PF keys (FSCAN) 9-132, 13-48
 - HELPMESSAGE text 9-131 to 9-132
 - PF keys in FIDEL 10-20
 - PF keys in Screen Painter 10-84, 10-87
 - PFKEY 10-21
 - field 10-23
 - query command 10-21
 - PFKeys dialog box 5-63
 - PFKeys help dialog box 5-83
 - PFkeys in Winform Painter 4-15, 5-14, 5-17
 - assigning to triggers 5-83
 - generating list in comments 5-29
 - PFnn parameter 10-22, 10-87
 - phrases 9-59
 - ON INVALID 9-99, 9-104
 - ON MATCH 9-59
 - ON MATCH/NOMATCH 9-61
 - ON NEXT 9-88 to 9-89
 - ON NONEXT 9-88 to 9-89
 - Pictorial View menu option in Winform Painter 5-29
 - PINK screen attribute (FIDEL) 10-30
 - setting in Screen Painter 10-93
 - placing text on form in Screen Painter 10-92
 - Pop-up option in Winform Properties dialog box 4-12, 5-38
 - positioning indexed fields 11-30
 - PRE_MATCH setting 7-111
 - Preferences menu option in Winform Painter 5-25
 - prefix area commands in FSCAN 13-7, 13-48
 - PREFIX parameter 9-190
 - COMBINE command 9-190, 9-194
 - presentation logic 1-6
 - WINFORM command 7-120

- previous command in FSCAN 13-37
- previous subcommand in SCAN 12-14
- procedures (Maintain) 1-5, 2-19, 6-8
 - beginning 3-5
 - blank lines 6-6
 - CALL command 2-19
 - comments 6-8
 - compiling 7-61, 7-89
 - data continuity 2-23
 - developing 1-5
 - ending 3-5
 - executing 1-7, 7-41
 - file extensions 5-10
 - flow of control 2-18
 - issuing 3-13
 - memory management 2-23
 - passing variables 2-20
 - remote 2-19
 - running 4-48, 7-102
 - storing 1-6
- process-driven development 1-5
- prompt for entry fields 4-24, 5-48
- PROMPT statement (MODIFY) 9-41, 9-43, 9-127
 - correcting field values 9-41, 9-47
 - FREEFORM statement 9-41, 9-50
 - MATCH statement 9-41, 9-49
 - NEXT statement 9-41, 9-49
 - typing ahead 9-41, 9-47
- properties (Winforms) 7-125
 - changing dynamically 7-125
- Properties menu option in Winform Painter 5-37
- Protected option for entry fields 4-24, 5-48
- Protected option in grid control 4-32, 5-56
- pull-down menus in Winform Painter 5-14

Q

- QQUIT command in FSCAN 13-4, 13-38, 13-45
- query commands 9-214
 - ? FDT and determining segment number 10-47
 - ? PFKEY 10-21
 - MODIFY 9-214
- QUIT command in FSCAN 13-38, 13-45
- QUIT statement (MODIFY) 9-41, 9-46
- QUIT subcommand in SCAN 12-14, 12-30
- quitting Screen Painter 10-97

R

- R prefix area command in FSCAN 13-30, 13-46, 13-48
- R value for ACCESS attribute 13-4
 - FSCAN 13-4
- radio button control 5-6, 5-72
- Radio Group menu option in Winform Painter 5-72
- Range option for validating entry fields 5-49
- reading data sources 1-4, 2-7, 2-30, 9-19
- read-only access in FSCAN 13-4
- read-write access in FSCAN 13-4
- REBUILD facility 11-4, 11-6
 - CHECK command 11-33
 - DATE NEW subcommand 11-41
 - EXTERNAL INDEX subcommand 11-26
 - INDEX subcommand 11-22
 - MDINDEX subcommand 11-47
 - MIGRATE subcommand 11-47
 - REBUILD subcommand 11-8
 - REORG subcommand 11-13
 - TIMESTAMP subcommand 11-39
- rebuilding data sources 11-5, 11-8

- REBUILDMSG parameter 11-7
- RECOMPILE command 7-89
- record-at-a-time processing 1-2
- records 2-3
 - copying 2-3
 - deleting 2-3
 - selecting 2-3, 7-59, 7-65
 - suppressing display of in SCAN 12-11
- RED screen attribute (FIDEL) 10-30
 - setting in Screen Painter 10-93
- REDEFINES command (MODIFY) 9-94
- REFRESH keyword in WINFORM command 7-120
- Regen menu option in Winform Painter 5-24
- REGION data source A-19
- rejection message syntax (MODIFY) 9-129
- relational expressions 8-22
- remote procedures 2-20
 - CALL command 2-20
 - MAINTAIN command 2-20
 - passing variables 2-20
- removing segments from data sources 11-13
- Rename menu option in Winform Painter 5-36
- renaming functions/cases in Winform Painter 5-81
- REORG subcommand of REBUILD facility 11-13
 - LOAD option 11-14
- REPEAT command (Maintain) 2-14, 7-90, 7-92, 7-96
 - branching out of loops 7-96
 - ending loops 7-96
 - establishing counters 7-94
 - nested loops 7-94 to 7-95
 - simple loops 7-92
 - UNTIL 7-93
 - WHILE 7-93
- REPEAT command (MODIFY) with FIDEL 10-61
- REPEAT statement (MODIFY) 9-159
 - GOTO ENDREPEAT phrase 9-159, 9-161
 - GOTO EXITREPEAT phrase 9-159, 9-161
 - MATCH and NEXT statements 9-159, 9-161
 - modification phase 9-159, 9-168
 - retrieving REPEAT 9-159, 9-163
 - stacking REPEAT 9-159, 9-161
 - syntax 9-159, 9-161
- REPEATCOUNT variable 9-159, 9-164
- repeating groups 9-41
 - fixed-format transaction data sources 9-31
 - PROMPT statement 9-41, 9-43
- repeating last command in FSCAN 13-37
- repeating last subcommand in SCAN 12-14
- REPLACE 13-31, 13-46
 - command in FSCAN 13-31, 13-46
 - keyword in USE command 11-31
 - subcommand in SCAN 12-13, 12-31
- REPLACE KEY command in FSCAN 13-35, 13-46
- replacing data using SCAN 12-13
- report procedures 2-49
- REPOSITION command (Maintain) 2-22, 3-6 to 3-7, 4-46, 7-97
- REPOSITION statement (MODIFY) 9-148 to 9-149, 9-171
- RESET command in FSCAN 13-30, 13-33, 13-46
- RESET keyword in GOTO command 7-44
- RESET keyword in WINFORM command 7-120
- RESET option 2-23
- Resize command in Winform Painter 4-27
- Resize menu option in Winform Painter 5-32
- resizing CRTFORMs 10-75
- resizing message area in FIDEL 10-77, 10-81

Index

- resizing screen in FIDEL 10-75
 - resizing Winforms 5-42
 - restrictions for FSCAN facility 13-2
 - RETAIN keyword 9-201
 - DEACTIVATE statement 9-206
 - return codes 9-107
 - FIND function 9-107, 9-109
 - LOOKUP function 9-107, 9-110, 9-115
 - return points 9-137
 - case logic 9-137, 9-139
 - RETURN setting for PFnn parameter 10-22
 - Return system action 5-89
 - return values for functions 7-19
 - RETURNS keyword in CASE command 7-16
 - reverse video fields (FIDEL) 10-25
 - dynamically changing 10-30
 - REVISE command 2-32, 3-11, 7-98, 7-100
 - REW screen attribute (FIDEL) 10-30
 - RIGHT command in FSCAN 13-15, 13-47
 - right justification 4-37, 5-61, 5-74
 - for button text 4-37, 5-61
 - for radio button text 5-74
 - Right system action 5-89
 - ROLLBACK command 2-36, 3-11, 7-101
 - DB2 data sources 2-50
 - ROLLBACK subcommand (MODIFY) 9-207, 9-209, 9-214 to 9-215
 - Absolute File Integrity 9-214
 - MATCH statement 9-214 to 9-215
 - NEXT statement 9-214 to 9-215
 - root segments 13-10
 - combined structures 9-190, 9-196
 - rounding numeric values 8-7
 - rulers in Winform Painter 5-14
 - RUN command 1-7, 7-63, 7-102, 9-198
 - MODIFY 9-198
 - running Maintain procedures 4-48, 7-102
 - RW value for ACCESS attribute in FSCAN 13-4
- ## S
-
- S0 segments with FSCAN 13-3
 - safeguarding transactions (MODIFY) 9-207
 - SALES data source A-11 to A-12
 - sample data sources A-1
 - CAR A-14 to A-16
 - CENTGL A-49
 - CENTSYSF A-49
 - Century Corp A-38 to A-41, A-43 to A-48
 - COURSES A-20
 - EDUCFILE A-9 to A-10
 - EMPLOYEE A-3, A-5 to A-6
 - FINANCE A-18
 - Gotham Grinds A-33 to A-37
 - ITEMS A-30
 - JOBFILE A-7 to A-8
 - LEDGER A-17
 - MOVIES A-29
 - REGION A-19
 - SALES A-11 to A-12
 - TRAINING A-23 to A-24
 - VIDEOTR2 A-31 to A-32
 - VideoTrk A-27 to A-28
 - Save As command in Winform Painter 4-19
 - Save As menu option in Winform Painter 5-21
 - SAVE command in FSCAN 13-37, 13-47
 - Save command in Winform Painter 4-19
 - Save menu option in Winform Painter 5-21
 - SAVE subcommand in SCAN 12-13, 12-33
 - SAY command 7-103 to 7-104
 - choosing between SAY and TYPE commands 7-104

- SBORDER parameter 4-3
- SCAN facility 12-1
 - adding segment instances 12-12
 - deleting fields and segments 12-13
 - entering 12-4
 - locating records 12-4
 - moving segment instances 12-13
 - quitting 12-14
 - saving changes 12-13
 - subcommand summary 12-15
- Scratch Pad Area 9-158 to 9-159, 9-165
 - initialization 9-171
- screen attributes (FIDEL) 10-25
 - changing in Screen Painter 10-93
 - dynamically changing 10-30
 - GRAY 10-93
 - setting in Screen Painter 10-93
 - specifying 10-25, 10-30
- Screen Painter 10-84
- SCREENINDEX variable with FIDEL 10-61
- scroll bars 5-4
- scrolling in FSCAN 13-14
- security and rebuilding data sources 11-5
- security in FSCAN facility 13-4
- SEG keyword in CRTFORM * command 10-47
- SEG keyword in FSCAN command 13-5
- SEG prefix with SAY command 7-104
- segments 7-82, 13-10
 - adding 11-13, 12-12
 - changing 11-13
 - current position in SCAN 12-4
 - DELETE command 7-37
 - deleting in SCAN 12-13
 - determining number 10-47
 - displaying descendant segments in FSCAN 13-23
 - segments (*continued*)
 - in FSCAN 13-3
 - INCLUDE command 7-51
 - moving in SCAN 12-13
 - NEXT command 7-82
 - removing 11-13
 - REVISE command 7-98
 - segments (Maintain) 5-75
 - adding fields to Winform 5-75
 - segments (MODIFY) 9-68, 9-71, 9-74, 9-76, 9-81, 9-83, 9-88, 9-92
 - SEGTYPE attribute 9-71
 - MODIFY 9-68, 9-71, 9-83 to 9-84, 9-148, 9-151
 - NEXT statement 9-88, 9-91
 - Select Master menu option in Winform Painter 5-24
 - SELECT SQL command equivalent (Maintain) 7-65
 - Set Colors dialog box 5-51
 - SET command 10-78 to 10-79
 - allocating space for variables 10-78
 - SET command 7-104
 - SET keyword in WINFORM command 7-120
 - SET parameters 10-33, 10-35 to 10-36
 - &CURSOR 10-33
 - &CURSORAT 10-36
 - SET parameters 7-104, 9-117
 - ALL 11-37
 - EXTTERM 9-117, 9-125, 10-25
 - FIELDNAME 9-190, 9-192
 - NODATA 7-124, 8-26
 - PATHCHECK environment variable 2-44, 2-47
 - PFnn 10-22, 10-87
 - PREMATCH 7-112
 - REBUILDMSG 11-7
 - SBORDER 4-3
 - setting in Maintain procedure with SYS_MGR.FOCSET 7-110

Index

- SET parameters (*continued*)
 - SHADOW 11-2, 12-2, 12-33
 - SORTLIB 11-22
 - TEXTFIELD 9-52, 9-54
 - with ON TABLE 7-105
- SET_PREMATCH setting 7-112
- set-based processing 1-2, 2-2 to 2-3, 2-30, 3-6
- SetStackMode 7-125
- setting brackets for Winform Painter 4-4, 5-26
- setting colors in Winform Painter 5-9
 - controls 5-51
 - WINFORM SET command 7-125
 - Winforms 5-41
- setting XEDIT as editor for Winform Painter 5-29
- shadow paging in FSCAN 13-4
- SHADOW parameter 11-2
 - using with SCAN 12-2, 12-33
- sharing data sources 2-44 to 2-45
- shortcut key for button control 4-37, 5-62
- short-path records and SCAN 12-11
- SHOW keyword in WINFORM command 4-42, 7-120
- SHOW option in FSCAN facility 13-5
- SHOW subcommand in SCAN 12-10, 12-33
- SHOW_ACTIVE keyword in WINFORM command 7-120
- SHOW_AND_EXIT keyword in WINFORM command 7-120
- SHOW_INACTIVE keyword in WINFORM command 7-120
- sibling segments (MODIFY) 9-71, 9-82
- Simultaneous Usage (SU) 9-8
- SINGLE command in FSCAN 13-26, 13-47
- single-precision decimal fields 9-26
- Size menu option in Winform Painter 5-42
- size of data sources for rebuilding 11-5
- Size to Fit option in Listbox dialog box 5-68
- sizing CRTFORMs 10-75
- sizing message area in FIDEL 10-77, 10-81
- SOME keyword 7-22, 7-31
 - in COMPUTE command 7-22
 - in DECLARE command 7-31
- SORT keyword in STACK SORT command 7-106
- sort libraries during indexing 11-22
- SORTHOLD statement 9-171, 9-187
- SORTIN and SORTOUT DDNAMEs 11-22, 11-26, 11-32, 11-44
- SORTLIB parameter 11-22
- source machines 9-8
- source segment in externally indexed data sources 11-30
- source stack 4-22, 4-31, 5-46, 5-57
- Source Stacks in Winform Properties dialog box 4-12, 5-39
- space for variables in -CRTFORMs 10-78
 - setting in Screen Painter 10-88, 10-93
- specifying lowercase in FIDEL 10-14
- specifying uppercase in FIDEL 10-14
- spot markers 10-12
 - <0x 10-12
 - <0x> 10-12
 - <n 10-12
 - <n> 10-12
 - /10-12
 - MODIFY 9-122
- STACK CLEAR command 4-46, 7-105

- stack commands 2-7
 - COMPUTE 2-15
 - creating 7-54
 - REPEAT 2-14
 - STACK CLEAR 7-105
 - STACK SORT 7-106
- stack editors 3-18
- STACK keyword in COPY command 7-27
- STACK prefix with SAY command 7-104
- STACK SORT command 2-14, 7-106
- stack variables 7-6
 - FocCount 7-42
 - FocIndex 7-43
- stacks 2-2, 2-4
 - anchor or segment 2-7
 - cells 2-2
 - clearing 2-17, 7-105
 - creating 2-6, 7-54
 - Current Area 2-16
 - defining 2-7
 - displaying 2-15, 4-29, 5-54, 5-58
 - editing 2-3, 2-11, 2-15, 7-6, 7-26
 - FocCount variable 2-12
 - FocIndex variable 2-12
 - grids 2-15, 3-18
 - index 2-12
 - looping through 2-14
 - naming 6-3
 - nondata source columns 7-55
 - rows 2-12, 3-15, 7-106
 - sorting 2-14
 - source columns 2-7
 - STACK SORT command 7-106
 - target segment 2-7
 - user-defined columns 2-10
 - virtual fields 7-25
- START statement 9-57
 - case logic requests 9-135, 9-137
 - rules 9-135
- statements 9-2 to 9-3, 9-5 to 9-6, 9-19
 - ACTIVATE 9-199, 9-201
 - CHECK 9-207
 - COMPUTE 9-93
 - CRTFORM 9-19
 - DATA 9-56
 - DEACTIVATE 9-206
 - ENDCASE 9-133
 - ENDREPEAT 9-159
 - EXIT 9-135
 - EXITREPEAT 9-159, 9-161
 - FIXFORM 9-20
 - FREEFORM 9-35
 - GETHOLD 9-171, 9-178
 - GOTO 9-137
 - LOG 9-126
 - MATCH 9-58
 - NEXT 9-88, 9-148 to 9-149
 - PERFORM 9-137, 9-139
 - PROMPT 9-41, 9-43
 - QUIT 9-16
 - REPEAT 9-159
 - REPOSITION 9-148 to 9-149, 9-171
 - SORTHOLD 9-171, 9-187
 - START 9-57
 - STOP 9-57
 - TYPE 9-117
 - VALIDATE 9-99, 9-105
- statistical variables 9-213
 - MODIFY 9-213
- STOP statement (MODIFY) 9-57
 - case logic requests 9-135
- strong concatenation 8-19
- structural integrity of data sources 11-8, 11-33
- structure diagrams A-1
- subclasses 2-57

subcommands 12-1, 12-6
 ? 12-14, 12-41
 AGAIN 12-14, 12-16
 BACK 12-17
 CHANGE 12-13, 12-18
 CRTFORM 12-20
 DELETE 12-21
 DISPLAY 12-10, 12-22
 END 12-14, 12-23
 FILE 12-14, 12-23
 INPUT 12-24
 JUMP 12-9, 12-25
 last 12-14
 LOCATE 12-8, 12-26
 MARK 12-27
 MOVE 12-28
 NEXT 12-8, 12-29
 period (.) after 12-11
 previous 12-14
 QUIT 12-14, 12-30
 REORG 11-13
 repeating last 12-14
 REPLACE 12-13, 12-31
 ROLLBACK 9-207, 9-209, 9-214
 SAVE 12-13, 12-33
 SHOW 12-10, 12-33
 summary 12-15
 TLOCATE 12-8, 12-35
 TOP 12-7, 12-37
 TYPE 12-10, 12-38
 UP 12-10, 12-39
 X 12-15, 12-40
 Y 12-15, 12-40

subscripted variables in WHERE expressions 7-68

subtraction in date expressions 8-13

subtraction operator 8-3 to 8-4

subtree in SCAN 12-4

superclasses 2-57

Switch To menu option in Winform Painter 5-34

synonyms 7-39

syntax summary (MODIFY) 9-217

SYS_MGR global object 7-107

SYS_MGR.DBMS_ERRORCODE command 7-107

SYS_MGR.ENGINE command 7-108

SYS_MGR.FOCSET command 7-110

SYS_MGR.GET_PREMATCH command 7-112

SYS_MGR.PRE_MATCH command 7-111

SYS_MGR.SET_PREMATCH command 7-112

system actions 5-83, 5-87
 Accept 5-89
 Backward 5-89
 Close 5-89
 Close_form 5-89
 Exit 5-89
 FieldLeft 5-89
 FieldRight 5-89
 Forward 5-89
 Left 5-89
 OnTop 5-89
 Return 5-89
 Right 5-89

System setup dialog box in Winform Painter 4-4, 5-26

system variables (Maintain) 7-6 to 7-8, 7-43
 FocCurrent 7-42
 FocError 7-42
 FocErrorRow 7-43
 FocFetch 7-43

T

T. prefix for turnaround fields in FIDEL 10-16
 dynamically changing from D. 10-30
 setting in Screen Painter 10-93

TABLEF command 11-36
 determining structural integrity of data sources 11-36

- TAG parameter 9-191
 - COMBINE 9-190, 9-193
 - qualifier for field names 9-190 to 9-191
- TAKES keyword in CASE command 7-16
- target segment in externally indexed data sources 11-30
- target segment stacks 2-7
- TED (text editor) 9-52
 - MODIFY 9-52, 9-55
 - setting as editor for Winform Painter 5-29
- temporary fields (MODIFY) 9-92 to 9-93
 - COMPUTE statement 9-93
 - VALIDATE statement 9-93, 9-99
- temporary fields and columns (Maintain) 2-10, 2-20
 - in data sources 7-25
 - redefining 2-20
- Terminal command in Winform Painter 4-4
- terminal configuration 2-28
- Terminal menu option in Winform Painter 5-26
- terminating Screen Painter 10-97
- Text command in Winform Painter 4-33
- text control 4-33, 5-53
- text editor (TED) 9-52
 - MODIFY 9-52
 - setting as editor for Winform Painter 5-29
- text expressions 8-19
- text fields 9-22, 9-26
 - editing with TED 9-52
 - FSCAN 13-3
 - MODIFY 9-26, 9-51 to 9-52
- Text menu option in Winform Painter 5-53
- THEN keyword 7-47, 8-25
 - in conditional expressions 8-25
 - in IF command 7-47
- time expressions 8-8
- time stamps for data sources 11-39
- TIMESTAMP subcommand of REBUILD facility 11-39
- title for Winforms 4-11, 5-38
- TLOCATE subcommand in SCAN 12-8, 12-35
- TO keyword in SAY command 7-103
- TOP 12-7, 12-37, 13-18, 13-47
 - command in FSCAN 13-18, 13-47
 - subcommand in SCAN 12-7, 12-37
- TOP case (MODIFY) 9-133
 - case logic 9-133
 - rules 9-135
- Top function 7-19
 - libraries and 7-64
- TRACE facility (MODIFY) 9-13, 9-157
- TRACEOFF setting 7-110
- TRACEON setting 7-110
- TRACEUSER setting 7-110
- TRAINING data source A-23 to A-24
- TRANS transaction type 9-129
- transaction data sources 7-70
- transaction integrity 1-2, 2-35, 2-44
 - across procedures 2-36
 - change verification 2-44, 2-46
 - locking 2-50
- transaction processing 2-34, 3-11
 - canceling 2-38
 - change verification 2-44
 - collecting values 3-8, 7-120
 - COMMIT command 2-36, 7-20
 - concurrent transactions 2-42
 - DELETE command 7-33
 - evaluating success 2-41

transaction processing (*continued*)
 INCLUDE command 7-50
 multiple data source types 2-36
 multiple DBMSs rollback 7-102
 reading multiple-path data sources 7-71
 REVISE command 7-98
 ROLLBACK command 7-101 to 7-102
 spanning procedures 2-36
 UPDATE command 7-116
 USE command 2-44, 2-48
 writing 2-3

transaction types 9-126, 9-129, 9-207
 INVALID 9-207
 NOMATCH 9-126
 TRANS 9-129

transaction variables (Maintain) 7-7 to 7-8
 FocCurrent 7-42
 FocError 7-42
 FocErrorRow 7-43
 FocFetch 7-43
 UPDATE command 7-118

transactions 2-3, 2-35
 processing 2-34, 3-11

transactions (MODIFY) 9-2, 9-126, 9-193, 9-207, 9-214
 combined structures 9-190, 9-193
 data sources 9-34
 safeguarding 9-207, 9-214
 types 9-126

triggers 2-29, 3-15, 4-41, 5-83
 defining 2-25
 processing 1-2

Triggers menu option in Winform Painter 5-41

TRUE value for logical expressions 8-22 to 8-23

truncating commands 13-7

turnaround fields (FIDEL) 10-14
 changing to display fields 10-30
 conditional and non-conditional 10-43
 designating in Screen Painter 10-93

TURQ screen attribute (FIDEL) 10-30

TYPE (Maintain) 7-113 to 7-114

TYPE command 7-113
 choosing between SAY and TYPE commands 7-104
 embedding spacing information 7-114
 including variables in a message 7-114
 justifying variables 7-115
 multi-line message strings 7-114
 truncating 7-115
 writing information to a file 7-115

TYPE command (MODIFY) with FIDEL 10-73

TYPE keyword 10-77, 10-81
 in -CRTFORM command 10-81
 in CRTFORM command 10-77

TYPE statement (MODIFY) 9-117
 case logic requests 9-135
 CRTFORMs 9-131
 customized log files 9-120
 embedding data fields 9-117, 9-120
 HELPMESSAGE attribute 9-117, 9-131
 MATCH statement 9-117 to 9-118
 NEXT statement 9-117 to 9-118
 ON INVALID phrase 9-99, 9-104
 screen attributes 9-125
 spot markers 9-122

TYPE subcommand in SCAN 12-10, 12-38

U

U value for ACCESS attribute in FSCAN 13-4

unconditional fields (MODIFY) 10-42

UNDE screen attribute (FIDEL) 10-30

underlining fields (FIDEL) 10-25
 dynamically changing 10-30

undoing changes in FSCAN 13-30, 13-33

UNHIDE keyword in WINFORM command 7-120

- unique segments (Maintain) 7-82
 - DELETE command 7-37
 - INCLUDE command 7-51
 - NEXT command 7-82
 - UPDATE 7-119
 - UPDATE command 7-119
- unique segments (MODIFY) 9-68, 9-71
 - case logic 9-148, 9-151
 - CONTINUE TO method 9-71 to 9-72, 9-88, 9-91
 - NEXT statement 9-88, 9-91
 - WITH-UNIQUES method 9-88, 9-92
- UNTIL keyword in REPEAT command 7-90
- UP
 - subcommand in SCAN 12-10, 12-39
- UPDATE action (MODIFY) 9-64, 9-66
 - descendant segments 9-71, 9-76
 - NEXT statement 9-148 to 9-149
- UPDATE command (Maintain) 2-32, 3-11, 4-42, 7-116 to 7-118
 - data source position 7-119
 - stacks and 7-118
 - transaction variables 7-118
 - unique segments 7-119
- update-only access in FSCAN 13-4
- updating data sources 7-98, 13-29
 - in FSCAN 13-29
 - REVISE command 7-98
 - UPDATE command 7-116
- updating field values 13-29
- UPPER keyword in [-]CRTFORM command 10-14
- uppercase in FIDEL 10-14
- Uppercase option 4-24, 4-32, 5-48, 5-56
- USE command 2-44, 2-48
 - activating external indexes 11-31
 - with FSCAN 13-2
- USER setting 7-110
- user-defined columns 2-10

V

- VALIDATE command (MODIFY) 10-42 to 10-43, 10-65, 10-70
 - with FIDEL 10-42 to 10-43, 10-65
 - with repeating groups 10-70
- VALIDATE statement (MODIFY) 9-99, 9-105
 - ACCEPT attribute 9-99
 - compiled calculations 9-93
 - DECODE function 9-99, 9-106
 - FIND function 9-109
 - LOOKUP function 9-107, 9-116
 - MATCH statement 9-99, 9-105
 - MISSING attribute 9-99, 9-105
 - multiple record processing 9-159, 9-169
 - NEXT statement 9-99, 9-105
 - ON INVALID phrase 9-99, 9-104
 - PROMPT statement 9-99
 - repeating groups 9-99
 - testing incoming data 9-99, 9-102
 - validating values from a list 9-99, 9-106
- validating user entries in Winforms 5-49
- variable-length character format 7-12
 - passing A0 variables to procedures 7-56
 - passing variables between procedures 7-12
- variables 7-22
 - &CURSOR in Dialogue Manager 10-33
 - assigning values with COMPUTE 7-22
 - CURSOR (MODIFY) 9-159, 9-167, 10-33
 - CURSORINDEX (MODIFY) 10-33, 10-65
 - declaring 7-22, 7-31
 - FOCURRENT 9-9
 - HOLDCOUNT 9-159, 9-164, 9-168
 - local vs. global 7-33
 - passing 2-20
 - PATHCHECK 2-47
 - REPEATCOUNT 9-159, 9-164
 - SCREENINDEX with FIDEL 10-61
 - subscripted in WHERE expressions 7-68
 - system 7-6 to 7-8

Index

- variables (Maintain) 4-20, 5-65
 - displaying in an entry field 4-20, 4-22, 5-45
 - generating list in comments 5-29
 - member 2-57
 - memory management 2-23
 - setting with a check box 5-65

verifying data sources 11-33

VIA keyword (MODIFY) 9-13, 9-56

- DATA statement 9-56

VIDEOTR2 data source A-31 to A-32

VideoTrk data source A-27 to A-28

virtual fields in data sources 7-25

W

W value for ACCESS attribute in FSCAN 13-4

WARNING setting 7-110

weak concatenation 8-19

Web deployment 1-2

WHERE keyword 7-27, 7-68, 7-72

- in COPY command 7-27
- in NEXT command 4-46, 7-66, 7-68, 7-72

WHILE keyword in REPEAT command 7-90

WHIT screen attribute (FIDEL) 10-30

- setting in Screen Painter 10-93

WIDTH keyword in CRTFORM command 10-75

WINFORM command 3-8, 4-42, 7-120

- flow of control 7-123

Winform Painter 1-2, 4-3, 5-1

- accessing 4-3, 5-11
- customizing 4-4, 5-25
- design screen 5-14
- dialog boxes 5-2
- exiting 4-19, 5-12
- files 5-10
- function keys 5-17
- generating code 4-42
- saving your work 4-19, 5-12

Winforms 2-3, 2-25, 5-33

- adding fields 5-77
- block mode 2-26, 2-28
- browsing stacks 3-15
- copying 5-35
- creating 5-34
- deleting 5-36
- designing 2-27
- event-driven development 2-25
- event-driven processing 2-29
- graphical user interface 1-2
- importing 5-22
- invoking procedures 2-26
- moving 5-43
- online help 2-26
- Painter 2-25
- parts 2-26
- properties 4-11, 5-37
- renaming 5-36
- resizing 5-42
- setting background color 5-41
- specifying system actions 5-87
- specifying triggers 5-85
- switching between 5-34
- terminal configuration 2-28
- title 4-11, 5-38
- triggers 2-25, 2-29
- validating user entries 5-49

WINFORMS file 1-6, 5-10

- CMS 1-6
- MVS 1-6
- presentation logic 1-6

WITH keyword in USE command 11-31

WITH-UNIQUES method 9-71, 9-74

- NEXT statement 9-88, 9-92

wizards 7-2

WRITE command 2-33

write-only access in FSCAN 13-4

writing to data sources 1-5, 2-7, 2-32

X

X subcommand in SCAN 12-15, 12-40

Y

Y subcommand in SCAN 12-15, 12-40

YELL screen attribute (FIDEL) 10-30
setting in Screen Painter 10-93

YRT keyword 7-22
in COMPUTE command 7-22
in DECLARE command 7-31

YRT parameter with REBUILD NEW DATE 11-41

YRTHRESH setting 7-110

YRTTHRESH parameter with REBUILD DATE NEW
11-41

Z

zoned decimal fields (MODIFY) 9-26
fixed-format data sources 9-26

Zoom menu option in Winform Painter 5-42

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