

WebFOCUS

WebFOCUS Modify Reference

WebFOCUS MODIFY Reference

Copyright © 1998 Information Builders, Inc.

Table of Contents

Data Management Facilities	3
MODIFY Facility	3
Structure of a MODIFY Request	4
Example of a MODIFY Request.....	5
Collecting Transaction Data.....	5
Collecting Transaction Data With FIXFORM.....	5
Collecting Transaction Data With FREEFORM	6
Determining Which Records Are Retrieved.....	7
MATCH.....	7
NEXT	7
Using Case Logic in a MODIFY Request.....	8
Example of a MODIFY Request That Uses Case Logic	8
Managing Your Data: Advanced Features	9
A Quick Review of the MODIFY Process	10
Maintaining Databases: MODIFY	11
Examples of MODIFY Processing.....	12
Adding Data to a Database	12
Updating Data in a Database	13
Deleting Data From a Database.....	13
Additional MODIFY Facilities	14
MODIFY Command Syntax	15
Executing MODIFY Requests	15
Executing a Request as a Stored Procedure.....	15
Executing MODIFY Requests Online.....	16
Other Ways of Maintaining FOCUS Files.....	17
The EMPLOYEE Database.....	17
Describing Incoming Data.....	18
Reading Fixed-format Data: The FIXFORM Statement.....	18
Reading in Comma-delimited Data: The FREEFORM Statement	27
Prompting for Data One Field at a Time: The PROMPT Statement.....	31
Entering Text Field Data	37
Specifying the Source of Data: The DATA Statement	39
Reading Selected Portions of Transaction Files: The START and STOP Statements.....	39
Modifying Data: MATCH and NEXT	40
The MATCH Statement.....	40
Adding, Updating, and Deleting Segment Instances	43
Performing Other Tasks Using MATCH.....	46

Modifying Segments in FOCUS Structures.....	47
Selecting the Instance After the Current Position: The NEXT Statement.....	57
Computations: COMPUTE and VALIDATE	59
Computing Values: The COMPUTE Statement	59
Validating Transaction Values: The VALIDATE Statement	63
Special MODIFY Functions	68
Messages: TYPE and LOG.....	74
Displaying Specific Messages: The TYPE Statement	74
Logging Transactions: The LOG Statement	79
Case Logic	82
Case Statement Syntax	82
Rules Governing Cases	83
Executing a Case at the Beginning of a Request Only: The START Case	84
Branching to Different Cases: The GOTO, PERFORM, and IF Statements.....	85
Case Logic Applications.....	92
Tracing Case Logic: The TRACE Facility.....	98
Sorting the Scratch Pad Area: SORTHOLD.....	99
Advanced MODIFY Facilities	100
Modifying Multiple Files in One Request: The COMBINE Command.....	100
Compiling MODIFY Requests: The COMPILE Command.....	104
Active and Inactive Fields	105
Protecting Against System Failures	110
Displaying MODIFY Request Logic: The ECHO Facility.....	111
Dialogue Manager Statistical Variables	113
MODIFY Query Commands	113
Managing MODIFY Transactions: COMMIT and ROLLBACK.....	114
MODIFY Syntax Summary.....	115
MODIFY Request Syntax.....	115
Transaction Statement Syntax	117
MATCH and NEXT Statement Actions.....	117
Index.....	119

Data Management Facilities

Topics:

- [MODIFY Facility](#)
- [Structure of a MODIFY Request](#)
- [Collecting Transaction Data](#)
- [Determining Which Records Are Retrieved](#)
- [Using Case Logic in a MODIFY Request](#)
- [Managing Your Data: Advanced Features](#)
- [A Quick Review of the MODIFY Process](#)

In order to make full use of a data file, you must have the means to add, change, or delete the information stored in it. The FOCUS facility that enables you to manage your data is called MODIFY. You use the MODIFY facility by creating a MODIFY request that contains commands that identify the data file, describe how new data values will be collected, and outline the maintenance actions to be performed.

These topics present the basic principles of the MODIFY facility. They introduce you to the basic terms, concepts, and techniques used in FOCUS to modify a data file. These topics discuss:

- The essential components of a simple MODIFY request, including the methods used to collect and manipulate data.
- Case logic, the method for creating complex, modular requests.

For more detailed information on the MODIFY facility, see *Maintaining Databases: MODIFY*.

MODIFY Facility

The MODIFY facility enables you to add, update, and delete values from data files. The following diagram represents the structure of the MODIFY process:

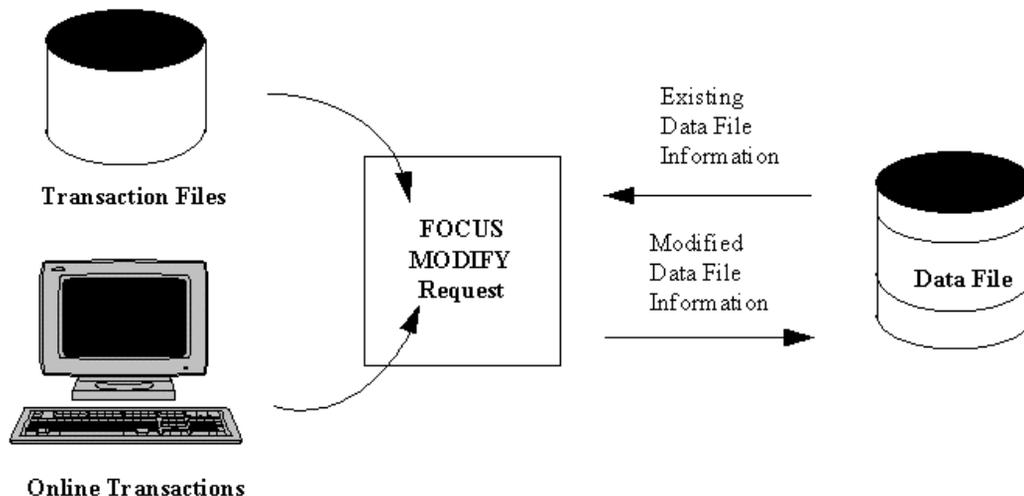


Figure 14-1. Overview of MODIFY Processing

Notice the key elements of MODIFY processing:

- Transactions** Transactions are collections of data values that will be used to change a data file. The values may be entered by users interactively, or they may be supplied from some other source, such as another data file.
- MODIFY Request** The FOCUS MODIFY request is the hub. It collects the transaction data, inspects the data file to determine whether an instance exists (that is, identifies the appropriate segment instance), and then follows the maintenance actions specified in the request.

Data file The data file stores the data that is managed by the MODIFY request.

Before you can manage data, the data file must exist. In FOCUS, a data file consists of one or more Master File Descriptions that describe the file (for some data file types, Access File Descriptions are also required) and the data files themselves if they already exist. For information about how to describe data, see Describing Data Files.

Structure of a MODIFY Request

A MODIFY request is made up of FOCUS MODIFY subcommands that describe the specific actions you wish to perform. The following example illustrates the basic structure of a MODIFY request. It is followed by an explanation of each of the elements:

```
1. MODIFY FILE filename
2. FIXFORM field1 field2 field3...
   FREEFORM field1 field2 field3...
   PROMPT field1 field2 field3...
3. MATCH field
   ON MATCH action
   ON NOMATCH action
```

(or)

```
   NEXT field
   ON NEXT action
   ON NONEXT action
4. CASE name
   MATCH field
   ON MATCH action
   ON NOMATCH action
   ENDCASE
5. DATA [ON ...]
6. END
```

1. MODIFY FILE filename invokes the MODIFY facility and identifies the data file to be modified.
2. In MODIFY the following commands describe how an application gathers its transaction data: FIXFORM, FREEFORM, or PROMPT.
 - FIXFORM reads the transaction values from a file. The file must be in fixed format, that is, each data value must occupy a fixed position in each transaction record.
 - FREEFORM reads transaction values from an external file. The file must be in comma-delimited format, that is, each data value must be separated by a comma (,), and each transaction record must be terminated by a comma and a dollar sign (,\$).
 - PROMPT prompts you for the transaction values, one field at a time. When you execute a request containing PROMPT commands, the request prompts you for each transaction field by displaying the field name and an equal sign (=). You can also specify your own prompt.
3. MATCH is one of the FOCUS commands that describes how FOCUS should use transaction data to modify a data file. MATCH determines which data file records are to be retrieved by comparing the transaction value for the named field (or fields) to the values stored for that field in the data file. If FOCUS locates an instance that matches, it retrieves the instance.
 - ON MATCH defines what actions to take if the MATCH command retrieves a matching instance from the data file. You can specify more than one ON MATCH command.
 - ON NOMATCH defines what actions to take if the MATCH command does not retrieve a matching instance from the data file.

NEXT is the other FOCUS command that, like MATCH, determines which data file records are to be retrieved. NEXT instructs FOCUS to select the next logical segment instance after the current instance and then requires an action to be performed.

- ON NEXT defines what actions to take if the NEXT command successfully retrieves another instance, from the data file. You can specify more than one ON NEXT command.

- * ON NONEXT defines what action to take if the NEXT command does not retrieve another instance from the data file (that is, if the end of the segment chain is reached). You can specify more than one ON NONEXT command.

Note: The basic MODIFY request should have at least one MATCH or NEXT.

4. You can divide MODIFY requests into executable units, or cases, to perform more than one action. For example, you might use one case to add new instances to the data file and another to delete instances, instead of having to write a separate MODIFY request for each action. Case logic enables you to develop requests in a modular way to make it easier to create sophisticated procedures that clearly specify and follow the flow of control.

CASE followed by a case name and ENDCASE enclose separate sections of the MODIFY request. Cases are named so you can switch processing to one case or another depending on a user selection, result of a test, etc. At the end of a case, processing can switch to another case, return to the beginning of the request, or exit the request.

5. The DATA command instructs FOCUS to read transaction values. It also indicates the source of the data. For example, DATA ON NEWEMP instructs FOCUS to read values from the file with the ddname NEWEMP.
6. END terminates the request when there is no more data to be processed. END is not used with PROMPT.

Example of a MODIFY Request

The following sample MODIFY request illustrates the elements discussed above:

```

MODIFY FILE EMPLOYEE
1. PROMPT EMP_ID PAY_DATE GROSS
2. MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH CONTINUE
3. MATCH PAY_DATE
   ON MATCH REJECT
   ON NOMATCH INCLUDE
DATA

```

This request adds a new monthly pay instance for each employee in the company. The request uses the EMPLOYEE data file. When the request is executed, it works as follows:

1. FOCUS prompts for transaction values for three fields in the EMPLOYEE file: EMP_ID, which is the keyfield in the EMPINFO root segment of the file, followed in turn by PAY_DATE, and GROSS (both of which are in the SALINFO segment of the file).
2. Once you interactively enter the values, the request matches the EMP_ID you entered against the EMP_IDs already in the data file. If a match is not found, the transaction is rejected. If a match is found, processing continues. The MATCH command matches one segment at a time; the next MATCH command will search for PAY_DATE.
3. The PAY_DATE you entered is matched against the PAY_DATEs stored in the file (for the specific EMP_ID already matched). If a match is found, the PAY_DATE is rejected and the transaction is not processed. If the PAY_DATE you entered does not match the value already in the data file, the transaction is accepted; a new instance of the segment containing PAY_DATE and GROSS is added to the record.

The next topics discuss the components of a MODIFY request further.

Collecting Transaction Data

To review, you can use any of the following MODIFY commands to describe how an application gathers its transaction data. The commands are: FIXFORM, FREEFORM, and PROMPT.

Collecting Transaction Data With FIXFORM

FIXFORM reads transaction values from a fixed format file, where every data value occupies a fixed position in each transaction record. FIXFORM reads the transaction values either from a file or from the data specified after the DATA command phrase.

The following is a portion of a typical MODIFY request using FIXFORM to read transaction values from a file:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/10
.
.
.
DATA ON NEWEMP
END
```

In this example, the command DATA ON NEWEMP tells FOCUS that the transaction data is stored in the file with ddname NEWEMP. When the request executes, FOCUS reads a set of transaction values for the fields EMP_ID, LAST_NAME, and FIRST_NAME into memory. Notice that the length of the transaction data is indicated after each field name. For example, EMP_ID/9 indicates that the length of transaction data for the field EMP_ID is fixed at nine bytes, LAST_NAME/10 indicates the length of transaction data is fixed at ten bytes; if the last name contains fewer than 10 characters, it must be padded with spaces to fill 10 bytes. You can also indicate format, for example, EMP_ID/D8 means the transaction data is a binary, double-precision floating point number.

FIXFORM also reads transaction data that is contained in the request itself. In this case, the transaction data immediately follows the DATA command, as shown in the following request:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/9
.
.
.
DATA
071382660STEVENS ALFRED
112847612SMITH MARY
END
```

FIXFORM can also be included in MATCH and NEXT logic. For example:

```
MATCH EMPLOYEE_ID
ON MATCH REJECT
ON NOMATCH FIXFORM ON COMPANY LAST_NAME/10 FIRST_NAME/9
ON NOMATCH INCLUDE
```

In this case, we are using the file COMPANY. When an employee ID is not found in the EMPLOYEE file, the information is taken from the COMPANY fixed format file.

Collecting Transaction Data With FREEFORM

FREEFORM reads transaction values from a comma-delimited file where every data value is separated by a comma (,), and where each transaction record is terminated by a comma and a dollar sign (,\$). FREEFORM reads the values either from a file or from the data included after the DATA command. FREEFORM provides the option of reading transaction values in the order specified in the request or in any order if the fields are identified in the transaction file, for example: LAST_NAME=Stevens, EMP_ID=112847612, FIRST_NAME=Smith,\$.

Look at the portion of a typical MODIFY request that uses the FREEFORM command to collect comma delimited transaction values from a sequential file:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID LAST_NAME FIRST_NAME
.
.
.
DATA ON NEWEMP1
END
```

In this example, the command DATA ON NEWEMP1 tells FOCUS that the transaction data is stored in the file NEWEMP1. When the request executes, FOCUS reads a set of transaction values for the fields EMP_ID, LAST_NAME, and FIRST_NAME from the sequential file NEWEMP1 into memory. FREEFORM does not require you to specify a format.

FREEFORM can also read transaction values into memory when those values are contained in the request itself. In this case, the transaction data immediately follows the DATA command, as shown in the following request:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID LAST_NAME FIRST_NAME
.
.
.
DATA
EMP_ID=071382660, LAST_NAME=STEVENS, FIRST_NAME=ALFRED, $
EMP_ID=112847612, LAST_NAME=SMITH, FIRST_NAME=MARY, $
END
```

FREEFORM can also be included in MATCH and NEXT logic. For example:

```
MATCH EMPLOYEE_ID
ON MATCH REJECT
ON NOMATCH FREEFORM ON COMPANY LAST_NAME FIRST_NAME
ON NOMATCH INCLUDE
```

In this MATCH command, when the employee ID is not found in the data file, the transaction data is taken from the COMPANY comma-delimited file.

Determining Which Records Are Retrieved

MATCH and NEXT commands in MODIFY requests determine which data file records are retrieved. Both MATCH and NEXT can be used to read transaction data, perform computations and validations, type messages, control case logic and multiple record processing, and activate and deactivate fields. NEXT can also be used to browse a data file chain.

MATCH

The MATCH command determines whether a specific segment instance exists in the data file. The ON MATCH and ON NOMATCH commands specify what actions are to be performed. The syntax is:

```
MATCH field
ON MATCH action-1...
ON NOMATCH action-2...
```

The ON MATCH and ON NOMATCH commands enable you to:

Reject the transaction values.	ON MATCH REJECT
Change the values of the named data file fields in the located segment instance.	ON MATCH UPDATE field1 field2...
Delete the instance and all children from the data file.	ON MATCH DELETE
Add a new instance and activate children in the data file.	ON NOMATCH INCLUDE
Reject the transaction values.	ON NOMATCH REJECT

For example, the following MATCH syntax adds a new instance (or instances) to a data file:

```
MATCH field
ON MATCH REJECT
ON NOMATCH INCLUDE
```

MATCH commands can also perform other functions like reading more transaction data, performing computations and validations, displaying messages on the screen or writing them to a log file, and switching processing to other parts of the request.

NEXT

NEXT selects the next logical segment instance after the current instance and uses ON NEXT/ON NONEXT commands to take particular actions. The syntax is:

```

NEXT field
  ON NEXT action-1...
  ON NONEXT action-2...

```

NEXT obeys the same rules as MATCH. NEXT uses ON NEXT/NONEXT to take specific actions. The ON NEXT and ON NONEXT commands enable you to:

Include a new instance after the current position.	ON NEXT INCLUDE
Reject the instance from the data file.	ON NEXT REJECT
Change the values of the named data file fields in the located segment instance.	ON NEXT UPDATE field1 field2...
Delete the instance and all children from the data file.	ON NEXT DELETE
Reject the transaction values.	ON NONEXT REJECT
Include a new instance at the end of the segment chain.	ON NONEXT INCLUDE

The NEXT command can be used to browse a chain of instances in the data file, display and modify the contents of unique segments, perform computations and validations, display messages on the screen or write them to a log file, and switch processing to other parts of the request.

Using Case Logic in a MODIFY Request

You can divide MODIFY requests into sections, or "cases," using Case Logic. This enables you to:

- Branch to a case from elsewhere in the request and process your transactions in different ways.
- Process multiple segment instances at one time.
- Perform calculations and store the results.
- Extend the use of the NEXT command to process segment chains.
- Display messages which contain transaction values.

The basic Case Logic syntax is

```

CASE casename
  .
  .
  .
ENDCASE

```

where:

casename Is a label of up to 12 characters that does not contain embedded blanks. CASE casename must be on a line by itself.

ENDCASE Is the command used to end a case. ENDCASE must be on a line by itself.

Example of a MODIFY Request That Uses Case Logic

The following request uses Case Logic to update employee salaries. If the salary is above \$50,000, the user will be asked to retype the value:

```

1. MODIFY FILE EMPLOYEE
   PROMPT EMP_ID CURR_SAL
3. IF CURR_SAL GT 50000 GOTO CONFIRM ELSE GOTO NEWSAL;
4. CASE NEWSAL
   MATCH EMP_ID
     ON MATCH UPDATE CURR_SAL
     ON NOMATCH REJECT
7. ENDCASE
8. CASE CONFIRM
9. TYPE
   "THE SALARY YOU ENTERED EXCEEDS $50,000"
   "PLEASE REENTER A NEW SALARY"
   PROMPT CURR_SAL
10. GOTO NEWSAL
12. ENDCASE
    DATA

```

Note:

- The request consists of three cases: the TOP Case (Lines 1-3), the CONFIRM Case (Lines 4-7), and the NEWSAL Case (Lines 8-12).

The TOP case is provided in FOCUS by default, and is assumed at the opening of the MODIFY request. Therefore, you do not need to name the case explicitly or to close it with the ENDCASE command, as required for other cases.

Notice that the NEWSAL and CONFIRM cases each begin with a case name and close with the ENDCASE command.

- Lines 3 and 10 incorporate the GOTO command. GOTO is used within cases to branch unconditionally to other cases. After the other case executes, control returns to the TOP case.

Two other commands can be used like GOTO. They are PERFORM and IF. PERFORM branches unconditionally to another case, but when the case called by PERFORM reaches ENDCASE, control returns to the statement following the PERFORM. IF branches to GOTO or to PERFORM, depending on the value of a logical expression.

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 topic describes them briefly.

Absolute File Integrity	Causes FOCUS to write changes to the data file to another section of the disk rather than overwriting the data file. If the request executes normally, the new section of the disk becomes part of the data file. If the system fails, the original database is preserved.
ACTIVATE [RETAIN] [MOVE]	The ACTIVATE command activates an inactive transaction field and moves the value of the corresponding data file field to the transaction field. If the transaction field is already active, no action is taken. The ACTIVATE RETAIN command activates the transaction field only. It declares a transaction field to be present so the transaction field can be used for matching, including, and updating. The ACTIVATE MOVE command moves the value of the corresponding data file field to the value of the transaction field and activates it.
DEACTIVATE [RETAIN]	Deactivates a transaction field and its corresponding data file 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. The DEACTIVATE RETAIN command deactivates the data file field, but does not change the value of the corresponding transaction field to blank or zero.
CHECK	Limits the number of transactions lost if the system fails when you are modifying a data file by identifying a checkpoint. CHECK activates the Checkpoint facility that enables FOCUS to write more frequently to the data file. (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 files in one MODIFY request.
COMMIT and ROLLBACK	Control the changes made to data files and protect the files 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 sink machine and source user IDs. COMMIT and ROLLBACK are used in lieu of CHECK.
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 files 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 file 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.
TYPE	Displays or stores messages in a separate file that you prepare.
VALIDATE	Enables you to reject transactions that contain unacceptable values.

A Quick Review of the MODIFY Process

Before concluding the overview, take a moment to review the three steps involved in MODIFY processing.

1. A transaction and its values are read into memory by the MODIFY request. Values can come from sequential files or from user entries on data entry screens or supplied in response to prompts. Values may also be included as part of the request itself.
2. The request selects a segment instance in the data file to modify by using either the MATCH command or using the NEXT command.
3. The request either adds, updates, or deletes data file values using transaction values, rejects transactions, or does nothing.

Now that you have a sense of what the MODIFY facility can do, you are ready to move on to Maintaining Databases: MODIFY, where the features discussed here are described in greater detail.

Maintaining Databases: MODIFY

Topics:

- [Examples of MODIFY Processing](#)
- [Additional MODIFY Facilities](#)
- [MODIFY Command Syntax](#)
- [Executing MODIFY Requests](#)
- [Other Ways of Maintaining FOCUS Files](#)
- [The EMPLOYEE Database](#)
- [Describing Incoming Data](#)
- [Modifying Data: MATCH and NEXT](#)
- [Computations: COMPUTE and VALIDATE](#)
- [Messages: TYPE and LOG](#)
- [Case Logic](#)
- [Advanced MODIFY Facilities](#)
- [MODIFY Syntax Summary](#)

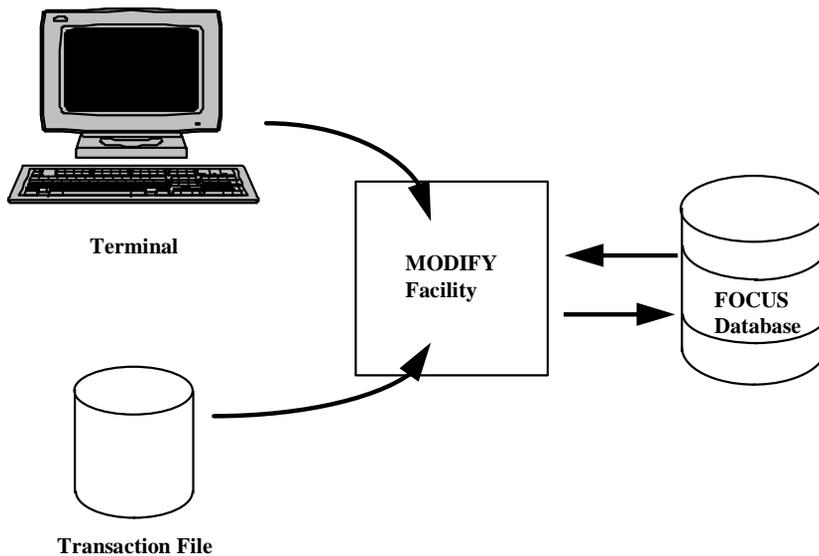
These topics describe how to maintain FOCUS-supported files using the FOCUS MODIFY facility. MODIFY requests can add, update, and delete data from FOCUS files, including HOLD files converted to FOCUS format (see FOCUS Utilities).

The MODIFY facility is also used to maintain data in relational structures. See the online *Relational Interface Reference* for details.

A MODIFY request processes a transaction in three steps:

1. It reads a transaction for incoming data values. Transactions can come from external files, responses 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 database.
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 describes the FIXFORM, FREEFORM, and PROMPT statements that describe transactions read by the request.

After it reads a transaction, the request selects a segment instance in the database to modify. It does this in either of two ways:

- It searches the database 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.

The request then either adds, updates, or deletes database values using the incoming values, or it rejects the transaction. The following examples show how this is done.

Examples of MODIFY Processing

The following topics provide examples of MODIFY processing that add, update and delete data from a database.

Adding Data to a Database

Updating Data in a Database

Deleting Data From a Database

Adding Data to a Database

The following sample MODIFY request adds new employee data to the EMPLOYEE database. When you execute 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 (or rejects it if it is duplicate information) to the database 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 database.
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 database for the employee ID that you entered.
4. If the ID is already in the database (that is, an ID in the database matches the ID you entered), the MATCH statement rejects your transaction.
5. If the ID is not yet in the database, the MATCH statement adds your transaction to the database.
6. The DATA statement begins prompting for data.

Updating Data in a Database

MODIFY requests can update data in a database, replacing database values with transaction (incoming data) values. The following sample request updates employee department assignments and salaries. When you execute the request, it reads the data from a separate file called EMPDEPT. Each record in the file 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 file:

```
* * * 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
    {MATCH EMP_ID
2.  {   ON NOMATCH REJECT
    {   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 file and describes the positions of the fields in each record.
2. The MATCH EMP_ID statement searches the database for the employee ID in each record. If the ID is not in the database, the request rejects the record. If the ID is in the database, the request replaces the DEPARTMENT and CURR_SAL values in the database with the values on the record.
3. The DATA statement indicates that the data is contained in the file EMPDEPT. EMPDEPT is the ddname that the file is allocated to and can be different from the system filename.
4. The END statement completes the request and initiates processing.

Deleting Data From a Database

This sample request deletes information on employees from the database. When you execute the request, it prompts you for an employee ID. When you enter the ID, it deletes all information relating to that employee from the database.

```

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 database. If the ID is in the database, the request deletes all information relating to the employee from the database.
3. The DATA statement begins prompting for data.

The above examples show how to add, update, and delete data from a database. Each request indicates the database it is modifying, the method of reading data, the transaction values it searches for in the database, and the actions it takes depending on whether the values are in the database or not. If it is reading a transaction file, the request must indicate the name of the file.

Additional MODIFY Facilities

You can also instruct the request to perform other tasks:

- Test transaction values to determine if they are acceptable. You do this using the VALIDATE statement.
- Perform calculations and store the results in either transaction or temporary fields. You do this using the COMPUTE statement.
- Display messages which contain values from transaction fields, temporary fields, or database fields. You do this using the TYPE statement.
- Record transactions processed by the request using the TYPE and LOG statements. 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 makes it possible for requests to offer the terminal operator selections and to process transactions in different ways.

FOCUS offers a variety of other advanced features that facilitate use of the MODIFY command in more complex applications.

- The COMBINE command for modifying multiple FOCUS files in one MODIFY request.
- The COMPILE command for translating MODIFY requests into compiled code ready for execution.
- The LOAD command places a MODIFY procedure into memory (see FOCUS Utilities for details).
- 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 files 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 databases.

The rest of these topics discuss the following:

- 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 files.
- A short description of the parts of the EMPLOYEE database most used in the examples.

MODIFY Command Syntax

The general syntax of the MODIFY command is

```
MODIFY FILE filename [ECHO ]
                    [TRACE]
.
.
statements
.
.
DATA [ON ddname ]
.
incoming data
.
.
[END]
```

where:

<code>MODIFY FILE</code>	Begins the request.
<code>filename</code>	Is the name of the FOCUS file you are modifying. This is the name of the Master File Description that points to the actual data file through either a USE statement for FOCUS files, a FILEDEF for ISAM, or an Access file for SQL databases, as determined by the SUFFIX= value in the Master File Description.
<code>ECHO</code>	Invokes the ECHO facility which displays the request logic
<code>TRACE</code>	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 and field activation status. Specify output destination (ddname HLIPRINT) with the appropriate FILEDEF statement. The default is TERMINAL.
<code>statements</code>	Are the MODIFY statements in the request. Each statement must begin on a separate line.
<code>DATA</code>	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.
<code>ON ddname</code>	Is a DATA statement parameter .
<code>incoming data</code>	Is the data you are using to modify the database if you are supplying the data in the request itself.
<code>END</code>	Concludes the request. Do not add this statement if the request contains PROMPT statements .

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 Managing Applications: Dialogue Manager). 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 file (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 file.
- The request encounters a GOTO EXIT statement.

Executing a Request as a Stored Procedure

To enter a MODIFY request as a stored procedure, type the request in a FOCEXEC file (FOCEXECs are discussed in Managing Applications: Dialogue Manager). 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 filename, 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 database.

`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
```

Executing MODIFY Requests Online

To execute a MODIFY request online, enter

```
MODIFY FILE filename
```

where:

`filename` Is the FOCUS name of the file you are modifying.

FOCUS responds with an echo of the filename, 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 file 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 file, or accept transaction records from the terminal (if the request contains FIXFORM or FREEFORM statements but does not specify the ddname of an external file).

If it accepts transaction records from the terminal, the request displays the following:

```
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 *Creating Tabular Reports: TABLE*.

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

TRANSACTIONS:  TOTAL=  3  ACCEPTED=  3  REJECTED=  0
SEGMENTS:     INPUT=  0  UPDATED=  3  DELETED=  0
```

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 topic.

Other Ways of Maintaining FOCUS Files

Although the MODIFY command is the primary method of maintaining FOCUS files, there are other facilities for changing data in FOCUS files:

- The Host Language Interface (HLI) allows you to maintain FOCUS databases from computer programs written in BAL, FORTRAN, COBOL, and PL/1. HLI is covered in the *Host Language Interface Users Manual*.
- 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 file. However, it cannot update key fields.

Note that although the FOCUS Report Writer can write reports from many kinds of non-FOCUS files (such as fixed format files), the MODIFY command maintains only FOCUS files, and with the proper interface, C-ISAM files, and SQL and Teradata tables.

The EMPLOYEE Database

The examples in these topics use the EMPLOYEE database, a database used to record employee information for a company. The Master File Description and the diagram of the entire file structure are shown in *Master File Descriptions and Diagrams*. Most of the examples use three segments in the EMPLOYEE file:

- 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 topic describes the statements that read and describe transactions. These are the FIXFORM, FREEFORM, and PROMPT statements.

To modify a database, 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 files 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 files and the request itself; some prompt users on terminals for data. The following table describes these statements:

Statement	Description
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.

Note: PROMPT, FREEFORM, and FIXFORM statements accept data that includes numbers expressed in scientific notation. For more information on the use of scientific notation in expressions, refer to Using Expressions, Functions, and Subroutines.

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. 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). You may also mix FREEFORM and PROMPT statements in one request.

If you are reading data from a file or user program, you must allocate the source of the data to a ddname.

Note: Do not begin any field used in a FIXFORM statement with Xn , where n is any numeric value. This applies to fields in the Master File Description and computed fields.

FOCUS allows the use of up to 3,072 fields in each MODIFY request. This total includes both database fields and temporary fields.

The following topics discuss several other features related to reading transactions:

- 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 file, the terminal, or a user program).
- The START and STOP statements that limit the request to reading a portion of the transaction file.
- The CHECK statement that controls the frequency that FOCUS writes transactions from buffer to disk.
- The Absolute File Integrity that uses a "shadow" or duplicate file to write completed transactions to, until the MODIFY procedure completes execution or a checkpoint is encountered.

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 files, 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 database. The data is stored in a sequential file called NEWEMP.

This is how the file appears on a text editor:

```
|.....+.....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 file 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 database, 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
```

FIXFORM Statement Syntax

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 ... fld-n 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 which does not appear in the Master File Description of the file 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 ...form-n` Are the formats of the incoming data fields, as described in FIXFORM Transaction Field Formats. The formats specify the format type (alphanumeric, integer, floating point, etc.) and the length of the field in bytes.

Note: No length is specified for the text field format which is variable in length. A FIXFORM statement can describe up to 12,288 bytes exclusive of repeating values.

To specify an alphanumeric format, just 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 database field regardless of internal format. Specifying the formats of binary, packed, and zoned transaction fields is discussed in FIXFORM Transaction Field Formats.

Remember that a transaction field can contain numbers and still be alphanumeric. If you display a transaction file 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 file 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 it 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 Description (named "master"). If you use this option, do not specify the fieldnames and formats in the FIXFORM statement itself. Use this option only if the Master File Description specifies a single segment file. All the fields in the Description specified by the FROM phrase must also appear in the Description 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 file called SALARY. Note that all the fields in the HOLD Master File Description must also appear in the SALARY Description.

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 Description or the FIXFORM format is C for conditional, or a combination of the two.

When duplicate fieldnames exist in a HOLD file, a MODIFY request that includes FIXFORM FROM HOLD should specify an AS name.

ALIAS

Indicates that the alias names from the Master File Description are to be used to build the FIXFORM statements.

Skipping 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`

instead of a field/format, 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 Moving Backward Through a Record.)

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 FIXFORM statement with `Xn`, where *n* is any numeric value. This applies to fields in the Master File Description and computed fields.

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 which 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 file is:

```
07138266023500.35
```

The statement interprets the record this way:

`EMP_ID/9` Reads the first nine bytes as the employee ID (071382660).

`X-9` Goes back nine bytes to the beginning of the record.

`EMP_CODE/3` Reads the first three bytes as the employee code (071).

X6 Moves forward six bytes.

CURR_SAL/8 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 Description, you must define the field with the COMPUTE statement before the FIXFORM statement.

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.

FIXFORM Transaction Field Formats

The FIXFORM statement enables you to read data in alphanumeric, numeric, date, and text formats. The syntax for each format is:

[A]n[YQMD] Specifies an alphanumeric character string *n* bytes long, where *n* is an integer. This is the default format. Date component options (YY, Y, Q, M, D) specify that the alphanumeric field is to be interpreted as a date; the options used and their sequence indicate the type of date. For more information about date format options, see [Describing Data Files](#).

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 if it is a date field.

F4 Specifies a four-byte binary floating point number.

D8 Specifies an eight-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 8 and *m* is an integer between 0 and 9. Date component options (YY, Y, Q, M, D) are included as necessary if it is a date field.

DATE Specifies a date field in four-byte integer format, to be copied to the database without date translation or validation. Date format fields can also be read without these restrictions by specifying alphanumeric, integer, or packed format.

TX Specifies a text format for 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 [Describing Data Files](#)).

Note:

- If more than one text field exists in the Master File Description, 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 file, 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 Description, 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 files, not the database. A transaction field can modify a database 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 database field.
- Alphanumeric format.
- Zoned format (if the database 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 database field), the request may terminate and generate an error message. To read such a transaction value into a database 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 database field. 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 file.
3. After the FIXFORM statement, use the COMPUTE statement to set a field with the same name as the database field equal to the value of the field you defined in Step 1. (**Note:** If the incoming field is numeric and the database field is alphanumeric, use the EDIT function to do this. The EDIT function is described in Using Expressions, Functions, and Subroutines.)

The following request reads a floating point field called FLOATSAL into the database 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 file.

Describing Date Fields

This topic 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 database date field, the corresponding data in the transaction file can be one of the following three types:

- A numeric date literal. For example, August 17 1989 can be represented in the transaction file 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 file as AUG 17 1989. The transaction field format must be An.

Note that all names of days and months in the transaction file must be in uppercase, even if the translation option is "t" or "tr." All abbreviated names of days and months in the transaction file 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 SAVEB and unformatted HOLD files. The date is stored as a four-byte integer representing the elapsed time since the standard FOCUS base date, as described in Describing Data Files. 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 Description from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE database. 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 four-byte integer representing the time elapsed from the standard FOCUS base date, as described in Describing Data Files. The transaction field will be copied directly to the database 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 database field's USAGE specification and edit options.

For example, if a database 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 database 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 Description; 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 database field), and you cannot convert one component to another (for example, you cannot convert a YM transaction field to a YQ database field). The only exceptions are the YY and Y date components, which can be substituted for each other.

Describing 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 which records the last 12 months of employees' monthly pay in the EMPLOYEE database. 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

```

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 database and the field remains inactive (active and inactive fields are discussed in Active and Inactive Fields).

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 file 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 database, 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 database, 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.

You may declare only alphanumeric fields as conditional. Therefore, if a binary, packed, or zoned field is not present in every record in the transaction file, then you must use Case Logic to process the file.

If you are adding segment instances, and several fields are conditional, values that are blank go into the new instances as follows:

- 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 Description (see Describing Data Files.).

If all fields in the segment are conditional and all values are blank, the segment will not be included.

- FIXFORM from Master assumes conditional input for all fields.

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 selectively read records only if a particular segment instance exists in the database (or confirmed not to be in the database).

For example, you design a MODIFY request that adds records of employees' monthly pay to the database:

```
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 files: EMPPAY and MONTHPAY. The EMPPAY file contains the employee IDs and the date each employee was paid. The MONTHPAY file contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

- The employee ID is recorded in the database. This is confirmed by the MATCH EMP_ID statement.
- The date the employee was paid has not yet been recorded in the database. This is confirmed by the MATCH PAY_DATE statement.

Once the request has confirmed this, it can read the monthly pay from the MONTHPAY file

```
ON NOMATCH FIXFORM ON MONTHPAY GROSS/7
```

and record it in the database:

```
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 files.

If the MODIFY request does not provide a statement reading transactions (FIXFORM, FREEFORM, or PROMPT), 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

```

FREEFORM Statement Syntax

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 file containing the incoming data. Use this option only when the DATA statement does not specify a ddname or specifies a ddname of a different file.

field-1 ... field-n 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. If you have FREEFORM fields that are not specified in the Master File Description, you must initiate them with a COMPUTE before you issue the FREEFORM command.

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 file.

If a MODIFY request has a FREEFORM statement, the statement must specify all the fields in the transaction file. If the transaction file 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 data entry statement in the request. Nevertheless, it is recommended that you 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 files.

If the Master File Description lists a date format with a translation option (see Describing Data Files.), you can type the date values in the transaction file 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 file 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 Description from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE database. 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 File

This topic discusses how MODIFY requests identify the values in comma-delimited files and determine what fields they belong to. (For more information on comma-delimited files, see Describing Data Files.) There are two types of values in comma-delimited files:

- Identified values are identified explicitly in the file.
- 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:

- Fieldnames or unique truncations of fieldnames. 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 fieldnames in the FREEFORM list.

Positional values exist by themselves without any identification in the file. For example:

```
SALES, 25000, $
```

The MODIFY request identifies positional values by the order of fieldnames specified in the FREEFORM statement list. If a record consists only of positional values, the request assigns the first fieldname in the list to the first value, the second fieldname 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 following 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 fieldnames 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 file 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 fieldnames declared in the Master File Description. If a record consists of only positional values, the request assigns the first fieldname in the Description to the first value, the second fieldname to the second value, and so on. For example, a transaction file 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 Description. The next value, MCHENRY, is the LAST_NAME field (the second field in the Description). DAVID becomes the FIRST_NAME field, the third field in the Description (the EMPLOYEE Master File Description is shown in Master File Descriptions and Diagrams).

If a record has both identified values 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 fieldnames that follow the name of the explicitly identified field in the Master File Description. For example, the transaction file 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 Description. Similarly, the request identifies PRODUCTION as the DEPARTMENT field.

Missing Values in Comma-delimited Files

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 database.

If you are adding segment instances to the database, 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 Description (see Describing Data Files.).

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 file 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 database. 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.

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 selectively read records if a particular segment instance exists in the database (or is confirmed not to be in the database).

For example, the following MODIFY request adds records of employees' monthly pay to the database:

```

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 files: EMPPAY and MONTHPAY. The EMPPAY file contains the employee IDs and the date each employee was paid. The MONTHPAY file contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

- The employee ID is recorded in the database. This is confirmed by the MATCH EMP_ID statement.
- The date the employee was paid has not yet been recorded in the database. This is confirmed by the MATCH PAY_DATE statement.

Once the request has confirmed this, it can read the monthly pay from the MONTHPAY file,

```
ON NOMATCH FREEFORM ON MONTHPAY GROSS
```

and record it in the database:

```
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.

PROMPT Statement Syntax

The syntax of the PROMPT statement is

```
PROMPT { field-1[.text.] field-2[.text.] ... field-n[.text.]
        { *
        }
      }
```

where:

`field-1 ... field-n` Are the names of the fields for which you are prompting. An asterisk (*) instead of fieldnames prompts for all fields described in the Master File Description 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 Description 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.

`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 06/19/1995 AT 14.38.27
  DATA FOR TRANSACTION 1

EMP_ID= >

```

where:

EMPLOYEE Is the system name of the database (in this case, the TSO name).

ON 06/19/1995

AT 14.38.27 Is the date and time that FOCUS opened the database: June 19, 1985 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 prompt for the EMP_ID field.

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 06/19/1995 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 you validate the field values after every prompt. This saves extra typing if one of the field values proves invalid. Validation tests are discussed in Validating Transaction Values: The VALIDATE Statement.

If the Master File Description lists a date format with a translation option (see Describing Data Files), 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 file was created.

For example, assume you change the format of the HIRE_DATE field in the EMPLOYEE Master File Description from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE database. 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) = > 86 jun 13

```

Note that you can also respond to the HIRE_DATE prompt with the value 860613.

Prompting 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 09/19/85 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 the following:

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 the following:

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.

Prompting Text

When you execute a request containing PROMPT statements, the request prompts you for each field by displaying the fieldname 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 fieldname 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 database:

```
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 in PROMPT

The following topics discuss special responses to prompts. They cover:

- Canceling a transaction.
- Ending execution.
- Correcting a field value.
- Typing ahead.
- Repeating the last response.
- Entering no data.
- Breaking out of repeating groups.

Canceling a Transaction

To cancel a transaction, enter a dollar sign \$ after any prompt. The request displays the following message

```
(FOC309) TRANSACTION INCOMPLETE:
      TRANS 1 REJ
```

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 database 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 fieldname 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 being prompted for.

`value-2 ... value-n` Are the values of fields not yet 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 execute 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 execute a request which 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 database.

If you are adding segments to the database, 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 Description (see Describing Data Files).

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 execute 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 database.

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 execute 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.

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 database 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 database. 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 database for the ID you entered. If it does not find it, it rejects the ID and prompts you for another ID. Only if it finds the ID in the database does it prompt you for a CURR_SAL value.

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. The only drawback to using it is that the transaction counter does not appear before the prompts. For example, a request might read employee ID numbers from a comma-delimited file, 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 file. Specify the file with the ON ddname option.
- Do not end the comma-delimited records with dollar signs (\$). End them with commas only (,).

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.

Entering Text Field Data

The following rules apply to text field data entry using the FOCUS text editor, 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 via the FOCUS Text Editor

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 the current release. 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 the current release.

In addition, FOCUS can preserve text fields exactly as entered into the database via ON MATCH/NOMATCH TED.

Defining a Text Field

The syntax for defining a text field in a Master File Description 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 fieldname.

nn Is the output display length in TABLE for the text field.

F Is used to format the text field for redisplay when the FOCUS text editor is called via 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 Description. Use of this determines whether text will display in the format in which it was entered.

For example, below is a Master File Description and the sample data that was entered into the field TXTFLD.

```
FILE=TEXT,SUFFIX=FOC
  SEGNAME=SEGA,SEGTYPE=S1
  FIELD=KEYFLD,,A1,$
  FIELD=TXTPFLD,,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 Description does not include the F option, the data will be redisplayed exactly as entered via the FOCUS text editor (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 will be omitted and the field will be 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.

The syntax of the DATA statement is

```
DATA [ ON ddname ]
```

where:

ON ddname Indicates the logical name assigned to the transaction files using FILEDEF.

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 database using the NEXT statement).

Reading Selected Portions of Transaction Files: The START and STOP Statements

MODIFY requests read and process transaction files from the first record to the last. The START statement signals requests to read starting from a particular record in the file. The STOP statement signals requests to stop reading at a particular record in the file. You may use START and STOP statements to process transaction files in sections, to resume processing a transaction file after a system crash, and to test a new request on a limited number of transactions.

The syntax for the START statement is

```
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 file (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. For example, if you want the request to read the file 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 file, 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 database. If a step is aborted for any reason, you can use the last backup to restore the file.

The following 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 determine which database 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 the following 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 in the previous example, the MATCH statement uses each of the two incoming data fields differently. It uses the EMP_ID field (specified after the word MATCH) to locate the segment instance (or to prove that it did not exist); it never alters the EMP_ID value in the segment. If it does locate the instance, it replaces 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. Remember from the introduction that FOCUS executes a MODIFY request for every transaction.

MATCH Statement Syntax

The syntax of the MATCH statement is

```
MATCH { * [KEYS] [SEG n]
      { [WITH-UNIQUES] field1 [field2 field3 ... field-n] }
      }
  ON MATCH action-1
  ON NOMATCH action-2
  [ON MATCH/NOMATCH action-3]
```

where:

WITH-UNIQUES Indicates that a parent and the unique child are to be considered one instance for the purpose of the MATCH.

field1 ... field-n Are the names of incoming data fields to be compared with similarly named database fields. The names may be full fieldnames, 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 database with incoming values, enter the following:

```
MATCH *
```

To compare the values of all key fields in the database with incoming values, enter the following:

```
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 FOCUS Utilities).

- `action-1` Is the action that is performed if the MATCH statement locates a segment instance with a data value matching the incoming data value (ON MATCH).
- `action-2` Is the action that is performed if the MATCH statement cannot locate a segment instance with a value matching the incoming data value (ON NOMATCH).
- `action-3` Is the action that is performed whether or not the MATCH statement locates a segment instance with a value matching the incoming data value (ON MATCH/NOMATCH).

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 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
```

Note: You may include many ON MATCH and ON NOMATCH phrases in one MATCH statement. ON MATCH phrases can precede or follow ON NOMATCH phrases.

The ON MATCH/NOMATCH Phrase

The MATCH statement has an ON MATCH/NOMATCH phrase that specifies an action to be taken whether or not the field value that the MATCH statement is searching for exists in the database.

You can specify the following actions in an ON MATCH/NOMATCH phrase:

- PROMPT
- GOTO
- IF
- ACTIVATE
- DEACTIVATE
- REPEAT
- HOLD

MATCH Statement Defaults

The following are defaults affecting the MATCH statement:

- If a MODIFY request has neither MATCH nor NEXT statements, it adds every transaction as a new segment instance, using incoming key field values to position the instance in the right segment. 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 database, the incoming data is in a file, and you know that there are no duplicate key values in the data.

The following request reads in data from a fixed-format file, ddname EMPDATA, to load in data into the segments EMPINFO and SALINFO in the EMPLOYEE database:

```
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:

```
ON MATCH CONTINUE
ON NOMATCH INCLUDE
```

- If a MATCH statement has an ON NOMATCH phrase but no ON MATCH phrase, the ON MATCH phrase defaults to:

```
ON MATCH CONTINUE
```

- If a MATCH statement has a MATCH phrase but no NOMATCH phrase, the ON NOMATCH phrase defaults to:

```
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 following 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.

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:

Phrase	Explanation
<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 database. Rather, it rejects the transaction.

- `ON MATCH UPDATE list` The instance exists in the segment. Therefore, the request changes the values of the database fields named in "list" to the values in the transaction.
- `ON MATCH DELETE` 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.
- `ON NOMATCH REJECT` The instance cannot be found in the segment. Therefore, it cannot be changed or deleted. The request rejects the transaction.

The following topics show how to construct MATCH statements that perform these functions.

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 Describing Data Files).

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 Sn segments, descending order for SHn 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 is harmless, but it does slow loading of data.

The following request adds new instances to the root segment of the EMPLOYEE database. 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 checks if ID 071382660 is in the segment. It is, 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 checks if ID 123456789 is in the segment. It is not, 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.

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 database 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 the following:

```
ON MATCH UPDATE * [SEG n]
```

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 071382662 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
```

Deleting Segment Instances

The syntax of the MATCH statement for deleting a segment instance is as follows:

```
MATCH keyfield
  ON MATCH DELETE
  ON NOMATCH REJECT
```

Note that the UPDATE action only updates fields when the transaction fields have values present.

The following 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 deletes all record of the employee from the database. This includes the employee's instance in the root segment and all descendant instances (such as pay dates, addresses, etc.).

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 reading incoming data, performing computations and validations, typing messages, controlling Case Logic and multiple record processing, and activating and deactivating 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.

Reading Data

The following actions read incoming data. They work just as FIXFORM, FREEFORM, and PROMPT statements:

<code>FIXFORM list</code>	Where <i>list</i> is a list of fields and formats. Reads in data from a fixed-format file.
<code>FREEFORM list</code>	Where <i>list</i> is a list of incoming data fields. Reads in data from a comma-delimited file.
<code>PROMPT list</code>	Prompts the user for data in fields named in <i>list</i> one field at a time.

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 "ddname."

Controlling Case Logic

The following actions control Case Logic.

<code>GOTO casename</code>	Branches to another case named by <i>casename</i> .
<code>PEFORM casename</code>	Branches to another case named by <i>casename</i> ,

then returns to the PERFORM.

```
IF expression  
GOTO case1  
[ELSE  
GOTO case2];
```

If the expression is true, the request branches to the case named by *case1*; otherwise the request branches to case named by *case2*.

Activating and Deactivating Fields

These actions activate and deactivate fields:

`ACTIVATE list` Activates fields named in *list*.

`DEACTIVATE list` Deactivates fields named in *list*.

Place these statements within a MATCH statement if you want to execute them only when the request can locate incoming values in the database (or confirm that incoming values are not in the database). This technique improves efficiency and makes the request logic more flexible.

Example of a Request Using MATCH Actions

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 database you are updating only contains the records of a small number of company employees, but the transaction file 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 file, whether or not the record will be accepted into the database. 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 database. 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 database. You may refer to database fields when using statements in MATCH and NEXT statements or after them. The database 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

The following topics discuss how the MATCH command modifies segments other than the root segment. They cover the following:

- Modifying unique segments.
- Modifying descendant segments.
- Modifying sibling segments (multi-path files).
- Modifying segments with no keys.
- Modifying segments with multiple keys.
- Using alternate views.

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.

The CONTINUE TO Method for Modifying Unique Segments

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:

<code>keyfield</code>	Is the key field of the parent segment instance.
<code>action-1</code>	Is the action the request performs if the parent instance cannot be found.
<code>u-field</code>	Is the name of any field in the unique child segment.
<code>action-2</code>	Is the action the request performs if a unique child instance exists.
<code>action-3</code>	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 and Performing Other Tasks Using MATCH. 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 file, 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.

The WITH-UNIQUES Method for Modifying Unique Segments

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:

- | | |
|-----------------------|---|
| <code>keyfield</code> | Is the key field in the parent segment. |
| <code>action1</code> | Is the action performed if the MATCH statement locates the parent instance. |
| <code>action2</code> | 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.

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 above.

This MODIFY request adds information on new employees, including information on direct-deposit bank accounts. If an employee is already recorded in the database, 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

```

For MATCH UPDATE fields, if a unique segment containing a field to be updated does not yet exist and data is present for that segment, that unique segment will be created and included automatically. For existing unique segments, the affected field is simply updated. Remember that only active fields in a MATCH UPDATE list will actually be updated (or included if the unique segment did not exist beforehand). If a MISSABLE field is set to MISSING and is to be updated in a MATCH UPDATE list, and the unique segment of the field does not yet exist, the field is set to MISSING, and the unique segment is include into the database.

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.

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 will illustrate this. The EMPLOYEE database 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

```

A sample 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.

If your request prompts for data using the PROMPT command, 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 above request as follows:

```

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 it, 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 database, the second MATCH statement searches the SALINFO segment for the pay date you entered. If it does not find it, 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 using a technique 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 above request 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 these topics 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.

Modifying FOCUS Structures of Three or More Levels

What has been said for two-level FOCUS structures is also 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 which 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
```

Modifying Sibling Segments (Multi-path Files)

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

```

Modifying 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 following form

```

MATCH { * [SEG n]
      { field-1 field-2 ... field-n }
      {
ON MATCH action-1
ON NOMATCH action-2

```

where:

`field-1 ... field-n` 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 FOCUS Utilities).

The difference between segment type S0 and blank is the way FOCUS adds new instances to the segments.

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 right after the current position (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 very hard 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 database that stores memos, called MEMO. The Master File Description 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 database:

```

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.

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.)

Modifying Segments With Multiple Keys

Segments may have multiple keys. These segments are types S_n or SH_n 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 Description. 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 Description:

```

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 file, 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.

Using 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 database.

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 file 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 via 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. For information on external files, see FOCUS Utilities.

This sample FOCUS data file, called BANK, contains information on bank accounts. The Master File Description 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=TRANSTYPE ,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 file.
- The TRANSSEG segment, child of ACCTSEG, contains information on individual bank account transactions: the transaction serial number (TRANSNUM), the type of transaction (TRANSTYPE 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 file, you 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 file structure, but not its parents. In the BANK file, you may modify the TRANSSEG segment using the above alternate file view but not the CUSTSEG segment.

The following request adds information on new bank account transactions to the database:

```
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. However, the NEXT statement is most often used for displaying database values.

Syntax of the NEXT Statement

The syntax of the NEXT statement is

```
NEXT field
    ON NEXT action-1
    ON NONEXT action-2
```

where:

- | | |
|-----------------------|---|
| <code>field</code> | Is any field in the segment whose instances are being selected. |
| <code>action-1</code> | Is the action the request takes if there is a next instance to select. |
| <code>action-2</code> | 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 and Performing Other Tasks Using MATCH). 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 database 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
```

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.

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 database 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.

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):

- The CONTINUE TO method.
- The WITH-UNIQUES method.

The CONTINUE TO Method for Displaying Unique Segments

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:

<code>field</code>	Is the first field in the parent instance.
<code>action-1</code>	Is the action the request performs if there are no more instances in the parent segment chain.
<code>u-field</code>	Is the name of any field in the unique child segment.
<code>action-2</code>	Is the action the request performs if the parent instance has a unique child instance.
<code>action-3</code>	Is the action the request performs if the parent instance does not have a unique child instance.

The WITH-UNIQUES Method for Displaying Unique Segments

The syntax of the WITH-UNIQUES method is

```
NEXT WITH-UNIQUES field
  ON NONEXT action1
  ON NEXT action2
```

where:

<code>field</code>	Is the name of any field in the parent segment.
<code>action1</code>	Is the action the request performs if there are no more instances in the chain.
<code>action2</code>	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, database 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.

Computing Values: The COMPUTE Statement

The COMPUTE statement allows you to modify incoming data field values and to define temporary fields.

A transaction file used to modify a database often does not contain the same data that is to go into the database fields. There are many reasons why this happens:

- The incoming data contains short codes representing the alphanumeric data that is to go into the database. 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 database. Using COMPUTE you can do several things:

- Translate codes into data to modify the database.
- Adjust the values of transaction fields.
- Define a data value or a series of data values to modify the database repeatedly.
- Calculate data values from other sources and use these new values to modify the database.

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 database 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 files or a terminal. These fields are specified by the FIXFORM, FREEFORM, and PROMPT statements. They remain incoming data fields even if their values are changed by COMPUTE statements.
- Database fields contain data stored in the database. Their fieldnames 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 database field, since its fieldname 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 file 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 fieldname exists in more than one segment, and that field must be redefined, the REDEFINES command should be used. For more information on the REDEFINES command see Creating Tabular Reports: TABLE.

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 the following:

- Database fields referred to in the request.
- Temporary fields created by COMPUTE and VALIDATE statements.
- Temporary fields created automatically by FOCUS, such as FOCURRENT for MODIFY requests run in Simultaneous Usage mode. FOCUS creates one FOCURRENT variable per request.

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 database field is read by a FIXFORM, FREEFORM, or PROMPT 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 execute and, should the MODIFY procedures be compiled, they will occupy more disk space.

For information on how to specify the old (prior to Release 5.5) logic and for a list calculations that default to the old compilation logic, see the topic on increasing the speed of DEFINE calculations in Creating Tabular Reports: TABLE.

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 database 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 database field values.

The following topics cover:

- The syntax of COMPUTE statements.
- Use of COMPUTE statements in MATCH and NEXT statements.
- Changing incoming data.
- Defining non-database transaction fields.

COMPUTE Statement Syntax

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 fieldnames). Fields can only modify database 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 Describing Data Files.

expression; Is any expression valid in a DEFINE or TABLE COMPUTE statement except for the LAST function (see Using Expressions, Functions, and Subroutines). In addition, you may use the FIND and LOOKUP functions.

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 Description.

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.

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 Creating Tabular Reports: TABLE.

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 database 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 database field values (as opposed to values in the transaction field with the same name), affix the D. prefix to the front of the fieldname.

Note that COMPUTE statements that follow a MATCH or NEXT statement may also perform calculations on database 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 database:

```
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
```

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 database. 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.

Defining Non-database Transaction Fields

If the names of incoming data fields are not listed in the Master File Description describing the database, you must define them to FOCUS before they are read in by a FIXFORM, FREEFORM, or PROMPT 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 Describing Data Files.

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.

Note: If the TOP CASE contains COMPUTE commands with expressions, as well as a command to read transaction values, the COMPUTE commands with expressions are executed after the read is performed. This applies to the TOP CASEs only.

For example, you want to record promotions to the MIS and Production Departments in the database. However, the transaction file you are working with lists the departments by code, not by name: 1 for MIS and 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 database.

Validating Transaction Values: The VALIDATE Statement

Most applications require that data be checked for accuracy before it is accepted into the database. 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 database. 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 databases, 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 database field values (D. fields).
- In MATCH and NEXT statements. VALIDATE statements here test the transaction depending on whether the MATCH or NEXT statement selected a particular segment instance. Expressions in these VALIDATE statements can use database 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.

The following topics describe:

- 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.

Note: MODIFY requests may also reject values that the ACCEPT attribute in the Master File Description declares to be acceptable. If this happens, the following message appears:

```
(FOC534) DATA VALUE IS NOT AMONG ACCEPTABLE VALUES FOR: fieldname.
```

See Describing Data Files. for further information.

VALIDATE Statement Syntax

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 fieldname or database fieldname 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 [Describing Data Files](#)). 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 (except for the LAST function. See [Using Expressions, Functions, and Subroutines](#)). Also, you may use the [LOOKUP](#) and [FIND](#) function. 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 [Creating Tabular Reports: TABLE](#).

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.

IF...THEN...ELSE Statements

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 [Creating Tabular Reports: TABLE](#)). Also note that all alphanumeric literals must be enclosed in single quotation marks.

Logical Expressions (MODIFY)

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 Using Expressions, Functions, and Subroutines. 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.

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.

The FIND Function

This function searches another FOCUS file 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 database, SALTEST is set to 0.

The following MODIFY request validates the DEPARTMENT and CURR_SAL fields:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT CURR_SAL
VALIDATE
    DEPTTEST = 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.

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 enables you to do two things:

- Branch to another case using Case Logic.
- Type a message.

The ON INVALID phrase immediately follows the validate statement. The syntax is

```
ON INVALID { GOTO casename
             { IF expression [THEN] [GOTO ] casename ELSE [GOTO ] casename;
             { PERFORM casename [PERFORM] [PERFORM]
             { TYPE[ON ddname]
             }
```

where:

GOTO casename Branches to another case called *casename*. If you omit this phrase, the request branches to the top of the request (the TOP case). GOTO also takes other options.

IF expression Branches to another case depending on the result of the expression.

PERFORM casename Branches to another case called *casename*, and execution then continues with the next statement after ON INVALID. PERFORM also takes other options.

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 file allocated to *ddname*.

The following 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 database 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 database field values, affix the D. prefix to the front of the fieldname.

Note that VALIDATE statements that follow a MATCH or NEXT statement may also use database 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 database:

```
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
```

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 Creating Tabular Reports: TABLE. These features test if an incoming field is present in the transaction or not and are especially useful when the transactions are on file.

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
```

Validating 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 Using Expressions, Functions, and Subroutines. This topic 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:

<code>field</code>	Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming fieldname or database fieldname for this name.
<code>fieldname</code>	Is the incoming data field being tested.
<code>code1 ...</code>	Is the list of possible values.
<code>result1</code>	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.
<code>ELSE</code>	Indicates what the temporary field is set to if the incoming field does not have a value on the list. This list may be up to 40 pairs of values if it appears in the request and up to 31,840 bytes long if it comes from a file.

For example, suppose you want to record promotions to various company departments in the database. 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
    DEPTEST = 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:

<code>field</code>	Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming or database fieldname for this name.
<code>infield</code>	Is the incoming field being tested.
<code>ddname</code>	Is the ddname of the file containing the list of possible values. The file may contain up to 31,840 bytes.
<code>m</code>	Is the value of <i>field</i> 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 MODIFY 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 files.

- The LOOKUP function, which tests for the existence of non-indexed values in cross-referenced FOCUS files 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 Using Expressions, Functions, and Subroutines.

Testing for the Existence of Indexed Values in FOCUS Files: The FIND Function

The FIND function verifies if an incoming data value is in a FOCUS database field, whether the field is in the database you are modifying or in another database. The function sets a temporary field to a non-zero value if the incoming value is in the database 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 database field. You can then test and branch on this field using Case Logic.

Note that the database field you are searching must be indexed, and that the FIND function does not work on files 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 database field containing values to be compared with the incoming data field. This field must be indexed. If the incoming field and the database field have the same name, you can omit this phrase.

file Is the name of the database.

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 database and 0 if the incoming value is in the database. Its syntax is

```
field = NOT FIND(infield [AS dbfield] IN file)
```

where *field*, *infield*, *dbfield*, and *file* are explained in the previous "where" list.

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 database. It then displays a message informing you whether it found the ID in the database.

```
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
```

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 database. If the field value is not in that database, 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 database, 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 database.

Reading Cross-referenced FOCUS Files: The LOOKUP Function

The LOOKUP function retrieves data values from cross-referenced files, both files cross-referenced statically in the Master File Description and files joined dynamically by the JOIN command. The LOOKUP function is necessary because unlike TABLE requests, MODIFY requests cannot read cross-referenced files freely. With the LOOKUP function, the requests can use the data in computations and in messages but cannot modify cross-referenced files; to modify more than one file in one request, use the COMBINE command.

The LOOKUP function can read cross-referenced segments that are linked directly to a segment in the host database (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 FOCUS Utilities).

The cross-referenced segment contains two fields of interest:

- The field containing the values you want. Alternatively, you can retrieve all of the fields in the segment at one time. The field, or your decision to retrieve all the segment's fields, is specified in the LOOKUP function.

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(s) from the first instance it finds.

The syntax of the LOOKUP function is

```
rcode = LOOKUP({field
                {SEG.segfield}
                })
```

where:

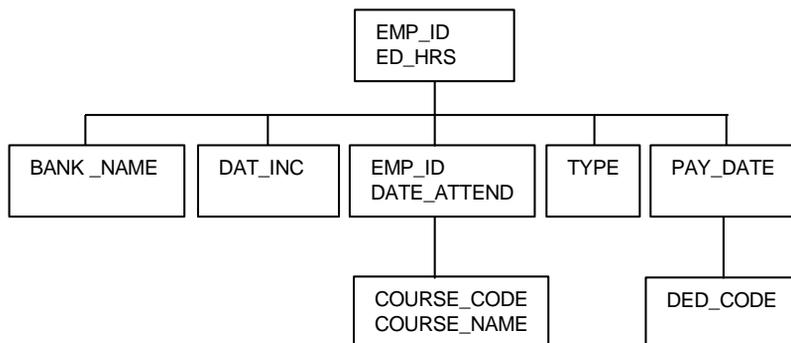
<code>rcode</code>	Is a variable you specify to receive a return code value. This value is 1 if the LOOKUP function can locate a cross-referenced segment instance, 0 if the function cannot.
<code>field</code>	Is the name of the field that you want to retrieve in the cross-referenced file. If the fieldname also exists in the host file, you must qualify it here.
<code>SEG.</code>	Specifies that all the fields in a segment will be retrieved. The segment is identified by the fieldname following this prefix.
<code>segfield</code>	Identifies the segment whose fields are being retrieved. It can be the name of any field in the desired segment.

Note that there may be no space between LOOKUP and the left parenthesis.

For example, suppose 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 file 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 database 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.
- 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 database.

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

```

LOOKUP Extended Syntax

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 file uses, enter the ? FDT command (see FOCUS Utilities).

Note that fields retrieved by the LOOKUP function do not require the D. prefix to be displayed in TYPE statements. FOCUS treats the field values as transaction values that are not active.

The extended syntax of the LOOKUP function is

```
rcode = LOOKUP(field ( {field          } {EQ} )
                  {SEG.segfield} {GE}
                  {                } {LE} )
```

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. If the fieldname also exists in the host file, you must qualify it here.

EQ, GE, LE These parameters specify the action the request takes if there is no cross-referenced segment instance corresponding to the host field value. The actions are:

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 next highest value of the cross-referenced field. The value of *rcode* is set to 2.

LE Causes the LOOKUP function to locate the instance with the next lowest value of the indexed field. The value of *rcode* is set to -2.

SEG. Specifies that all the fields in a segment will be retrieved. The segment is identified by the fieldname following this prefix.

segfield Identifies the segment whose fields are being retrieved. It can be the name of any field in the desired segment.

Note that there can be no space between LOOKUP and the left parenthesis.

The following table summarizes the value of *rcode*, depending on which instance the LOOKUP function locates:

Action	<i>rcode</i> 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

Figure 15-1. Values Returned by the LOOKUP Function

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 as follows:

```
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 and LOG

The following topics describe how MODIFY requests handle messages. There are four types:

- Messages written into requests.
- Messages indicating the rejection of transactions.
- 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.

Displaying Specific Messages: The TYPE Statement

The TYPE statement either displays on the terminal or stores in a sequential file messages that you prepare. The following topics cover:

- The syntax of the TYPE statement.
- Use of embedded data fields.
- Use of spot markers.

Note: Text fields cannot be used with the TYPE statement.

TYPE Statement Syntax

The syntax of the TYPE statement is

```
TYPE [AT START] [ON ddname]
    [AT END ]
```

```
"message"
["message"]
```

where:

AT START Displays a message at the beginning of execution only.

<code>AT END</code>	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 file, 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.
<code>ON ddname</code>	Writes the message to a sequential file that it FILEDEFs to <i>ddname</i> . 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.
<code>message</code>	Is any message. Enclose each line in double quotation marks (except when you want to display two lines as one line, as described Embedding Spot Markers.) 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, the following request warns the user when an employee ID that the user has entered is not in the database:

```
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. For example, the following 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"

```

Embedding Data Fields

You can embed data fields in the middle of messages. Embedded data fields are described in *Creating Tabular Reports: TABLE*. 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 database 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 database fields in messages. The database 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 database field, affix the prefix D. to the fieldname.

This TYPE phrase displays both the incoming value of CURR_SAL and the database 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 transaction into the file dname 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 database and records the ID and salary in the REJFILE file if the ID is not in the database. 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.

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 *Creating Tabular Reports: TABLE*. Some common spot markers are listed below (where *n* is an integer):

<n	Places text starting at the <i>n</i> th column.
<+n	Places text <i>n</i> columns to the right.
</n	Places text <i>n</i> lines down. Must be on a line by itself.
<0X	Positions the next character immediately to the right of the last character (skip zero columns). This is used when you have more than two lines between the double quotation marks in a FOCEXEC procedure that make up a single line of information. 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 in the Console window:

<p>THE DOLLAR SIGN IS IN COLUMN 40: \$ TEN SPACES ARE EMBEDDED IN THIS LINE</p> <p>THIS LINE SKIPS A LINE AND PROVIDES AN EXAMPLE OF THE USE OF A COLUMN MARKER</p>
--

Note: The spot marker to skip a line, </n, can appear on the same line with other text in a TYPE statement.

Sometimes, a line of text you want displayed cannot fit on one line within the TYPE command, either 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:

1. End the first line without an end quotation mark.
2. 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:

<pre>JOAN IRVINGEMP. #123764317 SALARY: \$27,000.00</pre>
--

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 follows:

<pre>SALARY UPDATE PROCEDURE WRITTEN JUNE 26, 1985 ENTER EACH EMPLOYEE ID AND SALARY AFTER THE PROMPTS</pre>
--

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 execute this request, the session looks like this:

```

> EMPLOYEE ON 10/10/85 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      = >

```

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.

Logging 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 database. 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.

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	All transactions processed by the request.
ACCEPT	Transactions accepted into the database.
DUPL	Transactions rejected because of an ON MATCH REJECT phrase (the transactions have field values that match those in the database).
NOMATCH	Transactions rejected because of an ON NOMATCH REJECT phrase (the transactions have field values that do not match values in the database).
INVALID	Transactions rejected because of data values that failed validation tests.
FORMAT	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).
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:

<code>FIXFORM</code>	The request stores the transactions in fixed format. Each <code>FIXFORM</code> statement retrieving data from the database logs one transaction. Each transaction consists of the fields defined by the <code>FIXFORM</code> statement plus the fields to the end of the physical record.
<code>FREEFORM</code>	The request stores the transactions in comma-delimited format. Each <code>FREEFORM</code> statement logs one transaction. Each transaction consists of one physical record delimited by a comma-dollar sign (<code>,,\$</code>). Note: Unless <code>FREEFORM</code> is explicitly included in the syntax, only the last line entered will be logged.
<code>PROMPT</code>	The request stores the transactions in comma-delimited format. Each <code>PROMPT</code> statement logs one transaction. Each transaction consists of data collected from the first <code>PROMPT</code> statement in the request to the <code>PROMPT</code> statement 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:

<code>FIXFORM</code> and <code>FREEFORM</code>	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 file (but is recorded as one line in the log file).
<code>PROMPT</code>	Define the record length as the sum of the lengths of the fields as defined by the <code>FORMAT</code> attribute (for example, a field having a format of <code>D12.2</code> has a length of 12), plus one byte for each field, plus one more byte.

The sample request below updates employee salaries and logs the transactions on five separate files. The original transaction file was stored in file `dname` `SALFILE`. Note the `VALIDATE` statement that checks if 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
```

There are five files specified in the `LOG` statements:

- The `ALLTRANS` file records all transactions.
- The `GOODTRAN` file records transactions accepted into the database.
- The `NOEMPL` file records transactions with employee IDs that do not exist in the database.
- The `BIGSAL` file records transactions with salaries that are too big (over \$50,000).
- The `BADFORM` file records transactions with salaries having invalid characters.

Controlling 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 database)

```
(FOC405 )TRANS n REJECTED DUPL: segment
```

where:

n Is the transaction number and *segment* is the database 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 database)

```
(FOC415) TRANS n REJECTED NOMATCH segment
```

where:

n Is the transaction number and *segment* is the database 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 via 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 via 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.

In addition, FOCUS displays the rejected transaction after each rejection message (except for format error transactions read in via PROMPT).

You may want to replace these messages with your own. To do so, use the TYPE statement. 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 in [Logging Transactions in Sequential Files](#).

ON *ddname* Logs the transaction in a file defined by *ddname*. This option is described in [Logging Transactions in Sequential Files](#).

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.

Case Logic

Case Logic allows you to branch to different parts of MODIFY requests during execution, enabling you to construct more complex MODIFY requests. For example, Case Logic requests can let the terminal operator choose 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.

Case Statement Syntax

Each case begins with the following 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 following 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 database. 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. (**Note:** 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 ELSE GOTO NEWSAL;

```

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 database.

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.
- There can be only one START case. If you include a START case, it must come after the TOP case.

- 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 or IF.
- 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

```
CASE REJECT
ON NOMATCH REJECT
ENDCASE
```

because ON NOMATCH REJECT is a phrase belonging to the MATCH statement.

- 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 statements. You can mix FREEFORM statements with PROMPT statements in one request.
- There is no limit to the number of cases you can use in a MODIFY request.
- If you use fields with D. and T. prefixes in TYPE statements, 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 following 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 branches unconditionally to another case. After the case executes, control returns to the TOP case.
- The PERFORM statement 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 branches to GOTO or PERFORM as above, depending on the value of a logical expression.

The GOTO Statement

GOTO statements unconditionally branch to another case. The syntax is

```

GOTO { TOP
      | ENDCASE
      | casename
      | variable
      | EXIT
    }

```

where:

<code>TOP</code>	Branches to the beginning of the TOP case.
<code>ENDCASE</code>	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.
<code>casename</code>	Branches to the beginning of the specified case.
<code>variable</code>	Branches to the beginning of the case whose name is the value of the temporary field <i>variable</i> . The temporary field must have a format of A12.
<code>EXIT</code>	Terminates the request. This is useful when you want to halt execution before the last transaction in a file 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.

The 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 {TOP  
        | ENDCASE  
        | casename  
        | variable  
        | EXIT  
        }
```

where:

<code>TOP</code>	Branches to the beginning of the TOP case. All return points are cleared and the procedure continues as if no PERFORM statement had executed.
<code>ENDCASE</code>	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.
<code>casename</code>	Branches to the beginning of a specified case.
<code>variable</code>	Branches to the beginning of the case whose name is the value of the temporary field <i>variable</i> . The temporary field must have a format of A12.
<code>EXIT</code>	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.

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 file 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
```

Suppose the file 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
```

Control goes to the beginning of the TOP case.
7. The TOP case prompts you for an employee ID and a salary.
8. You enter an ID Of 119329144 and a salary of \$65,000.
9. The PERFORM EMPCHECK statement branches to the EMPCHECK case. Since employee 119329144 is not listed in the EMPLIST file, the IF...GOTO TOP phrase branches to the TOP case.
10. The TOP case prompts you for an employee ID and a salary. You enter an ID of 071382660 and a salary of \$65,000.
11. The PERFORM EMPCHECK statement branches to the EMPCHECK case. Since employee 071382660 is listed in the EMPLIST file, control returns to the statement after the most recently executed PERFORM statement (PERFORM UPSAL).
12. 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).
13. The TYPE statement displays a message:

```
SALARY FOR EMPLOYEE 071382660 UPDATED
```

Control goes to the beginning of the TOP case.

The following rules apply to 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 the available memory. If memory runs out, FOCUS displays the following 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 and 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.

The IF Statement

The IF statement branches to another case depending on how an expression is evaluated. The syntax is

```

IF expr [THEN] {GOTO } {TOP } [ELSE {GOTO } {TOP } ]
                {PERFORM } {ENDCASE} | {PERFORM } {ENDCASE } |
                { } {case1 } | { } {case2 } |
                { } {var1 } | { } {var2 } |
                {EXIT } | {EXIT } |

```

where:

expr Is any logical expression legal in a DEFINE or COMPUTE IF statement (see Creating Tabular Reports: TABLE). 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 database fields unless they are used in or following MATCH or NEXT statements.

```

[THEN] {GOTO }
      {PERFORM }
      { }

```

Is the branch the request takes if the expression is true. 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*.

var1 Branches to the case whose name is contained in the temporary field *var1*. The temporary field must have a format of A12.

EXIT Terminates the request.

The word THEN is optional and is there to enhance readability.

```

ELSE {GOTO }
     {PERFORM }
     { }

```

Is the branch the request takes if the expression is not true. TOP, ENDCASE, case2, var2, and EXIT have the meanings explained above.

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. You place the inner IF phrase within parentheses following the THEN phrase. For example:

```

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 that 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 database.

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) 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 database fields in expressions. To do this, affix the D. prefix to the fieldname. 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 database value of CURR_SAL. The database 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, the following 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 database, 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), the GOTO, PERFORM and IF phrases in the statement are only executed when the MATCH statement modifies the last segment. For example, the following 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, 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

```

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, the following 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

The following topics discuss 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 children segments.
- Using Case Logic to offer user choices.
- Using Case Logic to process transaction files.
- Using Case Logic to process transactions based on the values of their fields.
- Using Case Logic to process transactions with bad values.

Looping Through a Segment Chain With the NEXT Statement

The NEXT statement modifies or displays the next segment instance after the current position in the database. Using Case Logic, you can use NEXT statements to process entire segment chains.

To display an entire segment chain, the request must branch back to the beginning of the NEXT statement until the whole chain has been displayed. 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 as follows:
 - 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.

Modifying Multiple Unique Segments

Modifying unique segments is described in Modifying Segments in FOCUS Structures. This topic 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 database, which records magazine subscribers, their mailing addresses, and expiration dates. The Master File Description 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 to determine if the subscriber has a magazine expiration date. If not, the request includes the new expiration date. If the subscriber has an expiration date then the request checks to determine if 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.

Using Case Logic to Offer User Selections

You can use Case Logic to offer users a selection of options. The request below offers a choice of 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

```

Using Case Logic to Process Transaction Files

You can use Case Logic to process records in a transaction file 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 and 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 file as being 90 bytes, the second FIXFORM statement reads a new record.
- The statement reads records from a different file 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 file.

If the next FIXFORM statement does not have the ON ddname option, it too reads another record.

Using 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

```

Using 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 following:

- 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 about 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 and 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 file 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.

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 10/04/85 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

```

Sorting the Scratch Pad Area: SORTHOLD

You can sort the contents of the Scratch Pad Area using any field or combination of fields in the Scratch Pad Area and 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.

`field2 ... 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 database key fields before entering a MATCH...UPDATE loop, to be sure that the transactions are in sequence with the database. Otherwise, you increase your execution time substantially. This FOCEXEC

performs this sort. It is issued after the records are displayed but before they are updated in the database.

Advanced MODIFY Facilities

The following topics discuss facilities that can assist you in using the MODIFY command:

- The COMBINE command for modifying multiple FOCUS files 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 files 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 databases.
- COMMIT and ROLLBACK subcommands for controlling changes made to FOCUS databases, and for protecting FOCUS files from system failures.

If you are operating in Simultaneous Usage mode (SU), please refer to the appropriate Simultaneous Usage manual.

Modifying Multiple Files in One Request: The COMBINE Command

The COMBINE command allows you to modify two or more FOCUS databases in the same MODIFY request. The command combines the logical structures of the FOCUS files into one structure while leaving the physical structures of the files untouched. This combined structure lasts for the duration of the FOCUS session, until you enter another COMBINE command, or until 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 64 segments.
- You can COMBINE files that come from different applications and have different DBA passwords. The only requirement is a valid password for each file. For more information refer to Providing File Security: DBA.
- Only the MODIFY and CHECK commands can process combined structures.
- If you are using Simultaneous Usage mode, all the files in the combined structure must either be all on the same sink machine or all in local mode.
- The differences between JOIN and COMBINE commands are discussed in Differences Between COMBINE and JOIN Commands.

COMBINE Command Syntax

Enter the COMBINE command at the FOCUS command level (at the FOCUS prompt). The syntax is

```
COMBINE [FILES] file-1 [PREFIX string-1] [TAG tag1] [AND]
file-2 [PREFIX string-2] [TAG tag2] [AND] ...
file-n [PREFIX string-n] [TAG tag3] AS name
```

where:

`file-1 ... file-n` Are the names of the data files you want to modify in the request. You may specify up to 16 files.

`PREFIX string` Is a parameter that enables you to refer to transaction fields with the prefix *string* of up to four characters. See Referring to Transaction Fields in Combined Structures: The PREFIX Parameter.

<code>TAG tag</code>	Is an alias for a filename of up to eight characters that enables you to refer to transaction fields. See Referring to Transaction Fields in Combined Structures: The TAG Parameter.
<code>AND</code>	Is an optional word to enhance readability.
<code>name</code>	Is the name of the combined structure that you will use in the MODIFY and CHECK commands. For example, if you name the combined structure EDJOB, begin the request with: <code>MODIFY FILE EDJOB</code>
<code>AS CLEAR</code>	Is the command that clears the combined structure that is currently in effect.

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.

For example, to combine files EDUCFILE and JOBFILE, enter:

```
COMBINE FILES EDUCFILE AND JOBFILE AS EDJOB
```

After entering this command, you can execute the following request. Notice that the statements pertaining to each file are placed in different cases (Case Logic is discussed in Case Logic). This clarifies the request logic and makes it easier to understand and clarify the request. The first case modifies the EDUCFILE database, and the second case modifies the JOBFILE database.

```
MODIFY FILE EDJOB
PROMPT COURSE_CODE COURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE

CASE EDUCFILE
MATCH COURSE_CODE
  ON MATCH REJECT
  ON MATCH GOTO JOBFILE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO JOBFILE
ENDCASE

CASE JOBFILE
MATCH JOBCODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
ENDCASE
DATA
```

Support for Long and Qualified Fieldnames

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 fieldnames. The syntax for this SET command is

```
SET FIELDNAME = { OLD      }
                 { NEW      }
                 { NOTRUNC }
```

where:

<code>OLD</code>	Specifies that 66-character and qualified fieldnames are not supported; the maximum length is 12 characters.
<code>NEW</code>	Specifies that 66-character and qualified fieldnames are supported; the maximum length is 66 characters. NEW is the default value.
<code>NOTRUNC</code>	Prevents unique truncations of fieldnames 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 FOCUS Utilities.

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 fieldnames, the fieldnames must be unique; that is, the transaction fieldnames in one file cannot appear in other files. Refer to any transaction fieldnames that are not unique by their aliases, or use the TAG parameter in the COMBINE command to assign a tag name to the files that share the transaction fieldnames.

When a file has a tag, refer to its transaction fieldnames by affixing the tag name to the beginning of each fieldname.

For example, this COMBINE command combines files EDUCFILE and JOBFILE into the structure EDJOB and assigns the tag AAA to all the transaction fields in the EDUCFILE file:

```
COMBINE FILES EDUCFILE TAG AAA AND JOBFILE AS EDJOB
```

When you create a request that modifies this structure, type the EDUCFILE fieldnames 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 fieldnames in the EDUCFILE file: COURSE_CODE and COURSE_NAME, making the new fieldnames AAA.COURSE_CODE and AAA.COURSE_NAME. Use these tagged fieldnames only in MODIFY requests that modify the combined structure.

Referring to Fields in Combined Structures: The PREFIX Parameter

For a MODIFY request to refer to fields in a combined structure by their fieldnames, the fieldnames must be unique so that there is no ambiguity in the request. That is, the fieldnames in one file cannot appear in other files. If there are fieldnames 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 files sharing the fieldnames.

When a file has a prefix, refer to its fieldnames with the prefix affixed to the beginning of each fieldname. The fieldname can be up to 66 characters in length. For example, this COMBINE command combines files EDUCFILE and JOBFILE into the structure EDJOB and assigns the prefix ED to all the fields in the EDUCFILE file:

```
COMBINE FILES EDUCFILE PREFIX ED JOBFILE AS EDJOB
```

When you enter a request modifying the structure, type the EDUCFILE fieldnames 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 fieldnames in the EDUCFILE file: COURSE_CODE and COURSE_NAME. The new fieldnames are EDCOURSE_CODE and EDCOURSE_NAME.

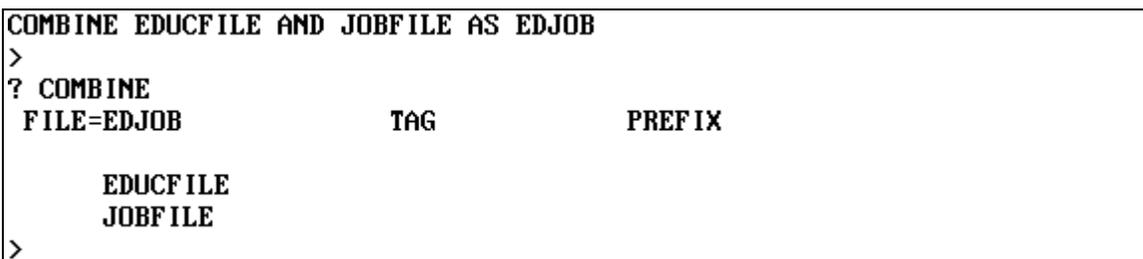
You use these prefixed fieldnames only in MODIFY requests modifying the combined structure. These prefixed fieldnames are not displayed by either the ?F query or the CHECK command.

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. For more information on loading files, see FOCUS Utilities.

How File Structures Are Combined

Combined structures start with a dummy root segment called SYSTEM, which becomes the parent of the root segments of the individual files. The SYSTEM segment contains no data. This is not an alternate view; the relationships between segments in each file remain the same.

The following figure shows how two files, EDUCFILE and JOBFILE, are combined into one structure. The first two diagrams represent the EDUCFILE and JOBFILE structures; the third diagram represents the combined structure. Note that the relationship between the two segments in each file does not change.



Fieldnames are considered duplicates when two or more fields are referenced with the same fieldname 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 external files. The COMBINE command combines FOCUS files only.

- The JOIN command can only join files with common fields. The COMBINE command can combine all FOCUS files.
- The JOIN command joins file structures together at segments with a common field. This can invert some of the segment relationships in the cross-referenced file (see alternate file view in Describing Data Files and Creating Tabular Reports: TABLE). The COMBINE command combines the file structures under a dummy root segment. Segment relationships remain intact.

The ? COMBINE Query

To display information on the combined structure currently in effect, enter the following command:

```
? 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 files that make up the combined structure.

tag-1 ... tag-n Are the tags attached to the fieldnames in the data file. These tags correspond to the aliases given to the file(s) in the combined structure.

prefix-1 ... prefix-n Are the prefixes attached to the fieldnames in the data file.

For example, when file EDUCFILE is combined with file JOBFIL, enter the following 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.

Compiling MODIFY Requests: The COMPILE Command

The COMPILE command translates a MODIFY request stored in a FOCEXEC 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 FOCEXEC where the request is stored.

module Is the name of the module. The default is the FOCEXEC name.

FOCEXEC names and module names are system dependent.

To execute a module, enter the following 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 FOCEXEC 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 database you are modifying is joined to another file (using the JOIN command) during compilation, it must be joined to the file 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.
- FOCEXECs 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

The following topics discuss active and inactive fields. When you execute 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 database.

If a MATCH or NEXT statement executes an UPDATE action, only active fields update the database. Database fields corresponding to the inactive incoming fields remain unchanged.

The following topics cover:

- When fields are active and inactive.
- [Activating fields with the ACTIVATE statement.](#)
- [Deactivating fields with the DEACTIVATE statement.](#)

When Fields Are Active and Inactive

A data field becomes active when:

- It is described in the Master File Description 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.
- 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.

Activating Fields: 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 database 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 database 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 Description (discussed in Creating Tabular Reports: TABLE).

The syntax of the ACTIVATE statement is

```
ACTIVATE [RETAIN] [SEG.]field1 field2 ... fieldn
        [MOVE ]
```

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 database field, even if the field was already active before the ACTIVATE statement.

field1 ... fieldn 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 fieldname. For example:

```
ACTIVATE SEG.SKILLS
```

The following 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 execute the request, the following happens:

1. The request reads the record:

```
EMP_ID=222333444, CURR_SAL=50000, ED_HRS=40, $
```

2. The following 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:      22223333444 (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 database. It therefore includes the record in the database 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 database 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 following 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 following 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 database ED_HRS value. The segment instance is now as follows:

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 following 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 as follows:

Transaction Record:

```
EMP_ID:      451123478 (inactive)
CURR_SAL:    50000 (active)
DEPARTMENT:  PRODUCTION (active)
ED_HRS:      50 (active)
```

11. The following 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 as follows:

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 database 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 as follows:

Database Segment Instance:

```
EMP_ID:      543729165
CURR_SAL:   16100.00
DEPARTMENT: PRODUCTION
ED_HRS:     50.00
```

The transaction record is now as follows:

Transaction Record:

```
EMP_ID:      543729165 (inactive)
CURR_SAL:   16100.00 (inactive)
DEPARTMENT: PRODUCTION (inactive)
ED_HRS:     50 (inactive)
```

Deactivating Fields: 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 Describing Data Files). It also deactivates the corresponding database field. The RETAIN option leaves the transaction value unchanged.

The syntax is

```
DEACTIVATE { [RETAIN] [SEG.]field-1 field-2 ... field-n
             { [RETAIN] ALL
             { COMPUTES
             }
```

where:

RETAIN Is an option that deactivates database fields but does not change the value of the corresponding transaction fields to blank or 0.

field-1 ... field-n 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 fieldname. For example:

```
DEACTIVATE SEG.SKILLS
```

ALL Is an option that deactivates all fields (including temporary fields).

COMPUTES Is an option that deactivates all temporary fields.

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:

- The Checkpoint facility.
- The Absolute File Integrity feature.
- The COMMIT and ROLLBACK subcommands.

Safeguarding Transactions: The Checkpoint Facility

The Checkpoint facility limits the number of transactions lost if the system fails when you are modifying a database. You can set checkpoints for transactions that are being read from a file, or from the terminal.

When a MODIFY request is executed, it does not write transactions to the database immediately; rather, it collects them in a buffer. When the buffer is full, FOCUS writes all transactions in the buffer to the database 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 database by using the checkpoint facility. When you activate the Checkpoint facility, FOCUS writes to the database 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 database when the buffer accumulates 500 transactions in CMS, or 1000 transactions in MVS.
OFF	Deactivates the Checkpoint facility.
n	Is an integer. Activates the Checkpoint facility. FOCUS writes to the database when the buffer accumulates <i>n</i> 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 database, 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 database by the MODIFY request that was executing when the system came down. You can have the request start processing the transaction file at the next transaction by using the START command.

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
```

Safeguarding FOCUS Files: Absolute File Integrity

The Absolute File integrity feature completely safeguards the integrity of a FOCUS database that you are modifying, even if the system experiences hardware or software failure. When you are using this feature, FOCUS does not overwrite the database 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 database. If the system fails, the original database is preserved.

Safeguarding Transactions: COMMIT and ROLLBACK Subcommands

To use COMMIT and ROLLBACK you must use Absolute File Integrity. Unlike the CHECK statement, COMMIT gives you control over the content of database changes and ROLLBACK enables you to cancel changes before they have been written to the database. 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.

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 several things:

- 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.
- The number of database 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 database.

When you execute the request, the request does not modify the database; rather, the ECHO facility displays the listing at the terminal. If ECHO is used against a non-FOCUS file, the word ECHO should appear on the second line, beneath MODIFY FILE *file*.

The ECHO facility can store the listing in a file rather than display it on the screen. To do this, use the FILEDEF command to allocate the file to ddname HLIPRINT. A record length of 80 bytes is sufficient.

The listing has the following form

```
MODIFY ECHO FACILITY
ECHO OF PROCEDURE: focexec
-----
CASE casename
-----
statements
      SEGMENT: segname
ON MATCH   ON NOMATCH
-----
match-actions  nomatch-actions
NUMBER OF DATABASE FIELDS:  n
TOTAL NUMBER OF FIELDS:    n
TOTAL SIZE OF FIELD AREAS:  n
```

where:

<code>focexec</code>	Is the name of the FOCEXEC that the request is stored in. If you entered the request from a terminal, this line is omitted.
<code>casename</code>	Is the name of the case, if the request uses Case Logic.
<code>statements</code>	Are the global (non-position related) MODIFY statements used. (Note: MATCH statements are shown separately.)
<code>segname</code>	Is the name of the segment being modified or used to establish a current position.
<code>match-actions</code>	Are actions taken on an ON MATCH or ON NEXT condition, including default actions.
<code>nomatch-actions</code>	Are actions taken on an ON NOMARCH or ON NONEXT condition, including default actions.
<code>n</code>	Is an integer.
<code>NUMBER OF DATABASE FIELDS</code>	Is the number of fields described by the Master File Description, including database fields in cross-referenced segments.
<code>TOTAL NUMBER OF FIELDS</code>	Is the sum of the number of database fields in the Master File description and temporary fields in the MODIFY request. This includes fields automatically created by FOCUS (these fields are listed in Computing Values: The COMPUTE Statement).
<code>TOTAL SIZE OF FIELD AREAS</code>	Is the sum of the sizes of database fields in the Master File Description 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 database. 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

```

When you execute 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 07/18/85 AT 10.48.21

MODIFY ECHO FACILITY
ECHO OF PROCEDURE: MOD76

CASE TOP

PROMPT
GOTO SALENTY

CASE SALENTY

SEGMENT: EMPINFO

MATCH

PROMPT
VALIDATE
INVALID TYPE
INVALID GOTO SALENTY
UPDATE

NOMATCH

REJECT

END OF ECHO:

NUMBER OF DATABASE FIELDS : 34
TOTAL NUMBER OF FIELDS : 36
TOTAL SIZE OF FIELD AREAS : 383

Dialogue Manager Statistical Variables

After you execute 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 FOCEXECs after execution (that is, after the Dialogue Manager -RUN control statement).

The variables that pertain to MODIFY requests are described in the following table:

Variable	Description
&TRANS	Number of transactions processed.
&ACCEPTS	Number of transactions accepted into the database.
&INPUT	Number of segment instances added to the database.
&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 FOCEXECs, see *Managing Applications: Dialogue Manager*.

MODIFY Query Commands

Four query commands display information regarding the MODIFY command and the maintenance of FOCUS files:

Command	Action
? COMBINE	Displays information on combined structures (see Modifying Multiple Files in One Request: The COMBINE Command).
? FDT	Displays information regarding the physical attributes of FOCUS files (see FOCUS Utilities).
? FILE	Displays information regarding the number of segment instances in FOCUS files and the dates and times the files were last modified (see FOCUS Utilities).
? STAT	Displays statistics regarding the last execution of a request (see FOCUS Utilities).

Managing MODIFY Transactions: COMMIT and ROLLBACK

COMMIT and ROLLBACK are two MODIFY subcommands. COMMIT gives you control over the content of database 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 database.

The MODIFY CHECK statement only enables you to control the number of transactions that must occur before changes are written to the database. When using CHECK, you cannot change the checkpoint setting once the MODIFY request begins execution. Similarly, changes cannot be canceled (see Protecting Against System Failures 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 databases, except those that have an A6 filemode. Absolute File Integrity for databases with an A6 filemode is provided by the FOCUS Shadow Writing Facility. Absolute File Integrity for databases in MVS/TSO is provided solely by the FOCUS Shadow Writing Facility. See Providing File Security: DBA and FOCUS Utilities for information on Absolute File Integrity and the SET SHADOW command.

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, because they will take precedence over CHECK processing.

Coding With COMMIT and ROLLBACK

COMMIT and ROLLBACK both process a logical transaction. A logical transaction is a group of database changes in the MODIFY environment that you want to treat as one. 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 database. COMMIT can be coded as a global subcommand or as part of MATCH or NEXT logic. The possible MATCH and NEXT statements are as shown:

```

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 database. 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 as follows:

```

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 or lack of disk space), no changes are made to the database. In this way, COMMIT is an all-or-nothing feature that ensures database 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 database when the user chooses to commit the changes or when the procedure is terminated without issuing a ROLLBACK subcommand.

MODIFY Syntax Summary

The following topics present a summary of MODIFY command syntax. The syntax of each statement is shown as part of a MODIFY request. The rest of the topics describe:

- The syntax of the transaction statements FIXFORM, FREEFORM, and PROMPT.
- 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

REPEAT { *          } [TIMES] [MAX maximum] [NOHOLD]
       { number    }
       {           }

```

```

statements
HOLD [SEG.]field [field ... field]
ENDREPEAT

ACTIVATE [RETAIN] [SEG.]field ... field
        [MOVE ]

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 } [TOP ] [ELSE {GOTO } {TOP } ]
                    {PERFORM} {ENDCASE | ENDCASE |
                    | ENDREPEAT | {PERFORM} | ENDREPEAT |
                    | casename | | casename |
                    | variable | | variable |
                    | EXIT | | 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 } [ON ddname] [MSG {ON } ]
   |ACCEPTS| | {OFF} |
   |DUPL | | { } |
   |NOMATCH}
   |INVALID|
   |FORMAT |
   | }

CHECK {ON }
      |OFF}
      |n }

START n

STOP n

```

```
DATA [ON ddname ]
      [VIA progame]
```

```
[END]
```

Transaction Statement Syntax

The following is the syntax for three transaction statements: FIXFORM, FREEFORM, and PROMPT.

The syntax of the FIXFORM statement is as follows:

```
FIXFORM {FROM master
        { [ON ddname] field/[C]format field/[C]format ... [Xn] [X-n] }
        }
```

The syntax of the FREEFORM statement is as follows:

```
FREEFORM [ON ddname] field field field ...
```

The syntax of the PROMPT statement is as follows:

```
PROMPT { *
        { field[.text.] field[,test,] . . . }
        }
```

MATCH and NEXT Statement Actions

This topic 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)
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
TYPE
UPDATE (ON MATCH and ON NEXT only)
VALIDATE
```

The following actions can be used in ON MATCH/NOMATCH phrases:

```
ACTIVATE
COMMIT
DEACTIVATE
GOTO
HOLD
IF
PERFORM
PROMPT
ROLLBACK
TED
```

The following actions can be used in ON INVALID phrases:

```
GOTO
PERFORM
TYPE
```


Index

A

Absolute File Integrity	9, 110
ACTIVATE	9
ACTIVATE statement	106
Active fields	105
Adding segment instances.....	27, 44
Advanced MODIFY facilities	100
Alternate file views	56
AT END.....	74, 75

B

Base date	24
Branching	89
Breaking out of repeating groups	36

C

Calculations.....	61
Case logic.....	5, 8, 25, 46, 82, 88, 91, 92, 94, 95, 96, 97, 98
Changing incoming data	63
Checkpoint facility	9, 110
COMBINE	10, 100, 101, 102, 103, 104
Comma-delimited files	28, 29, 30
COMMIT	10, 110, 111, 114, 115
COMPILE	10, 104
COMPUTE	10, 46, 59, 60, 61, 62, 63, 68
Conditional fields	26
CONTINUE TO	48, 49, 59
Correcting field values	35
CRTFORM	105, 106

D

Data management	3
DATA statement	39
Date fields	24
DEACTIVATE	9, 109
Deactivating fields	47

DECODE function	66, 67, 68
DELETE.....	45, 46
Deleting segment instances	43, 45
Describing date fields	24
Describing incoming data	18
Dialogue Manager.....	113
Displaying text fields	38
Duplicate fieldnames	21
E	
ECHO.....	10, 111, 112, 113
Embedded data	76
Embedding data fields	76
EMPLOYEE database	17
END.....	31, 32, 35
Entering no data	36
Entering text field data.....	37
EXIT	84
F	
Field formats	23, 62
FIND function	66, 69
FIXFORM	5, 6, 18, 19, 20, 21, 22, 27
FREEFORM.....	4, 6, 7, 27, 28, 29, 30, 31, 37
G	
GOTO statement.....	85, 89
H	
Host Language Interface.....	17
I	
IF statement	85, 87, 88
Inactive fields	105
INCLUDE	44, 49, 50
Integer fields	24
J	
JOIN	103, 104

K

Key fields 41, 54, 55

L

LOG 74, 79

Logging transactions 79, 81

Logical expressions 65, 66

LOOKUP function 69, 70, 71, 72, 73, 74

M

MATCH 4, 5, 7, 30, 31, 36, 37, 40, 41, 42, 43, 44, 46, 47, 48, 49, 53, 54, 57, 62, 67, 90, 91, 115, 117

Messages 74

Missing data 67

Missing values 30

MODIFY 3, 4, 5, 9, 10, 11, 12, 17, 18, 40, 59, 60, 61, 68, 69, 74, 82, 83, 84, 100

MODIFY command 15

MODIFY facilities 14

MODIFY processing 12

MODIFY requests 5, 8, 15

MODIFY syntax 115

Modifying descendant segments 50

Modifying segments 47

Modifying unique segments 47

Moving backward through a record 21

N

NEXT 7, 8, 30, 36, 40, 57, 58, 59, 62, 67, 92, 93, 105, 117

Non-Case Logic requests 58

O

ON ddname option 20

ON INVALID 66, 67, 91

ON MATCH 41, 42, 43, 44

ON NOMATCH 7

P

PERFORM 85, 86, 87, 90

Positioning
text 77, 78

PREFIX	102, 103
PROMPT	4, 5, 31, 32, 33, 34, 35, 36, 37
Protecting against system failures.....	110
Q	
Qualified fieldnames	101
Query commands.....	113
QUIT.....	32
R	
Reading data	46
Rejection messages	80, 81, 82
Repeating a previous response	36
Repeating groups	25, 33, 34
RETAIN.....	107, 109
ROLLBACK.....	110, 111, 114, 115
Rules governing cases.....	83
RUN.....	105
S	
Scratch Pad Area	99
Segments with multiple keys	55
Segments with no keys	54
Selecting instances	58
Sibling segments	53
Skipping columns	21
SORTHOLD	99, 100
Spot markers.....	77, 78
START case.....	83, 84, 85
START statement.....	39, 40
Statistical variables	113
STOP statement.....	39
Syntax summary	115
T	
TAG.....	101, 102
Temporary fields.....	59, 60, 61
Text fields	22, 38
TEXTFIELD.....	38

TOP	83, 84
TRACE facility.....	98
Transaction data	5, 6, 7
Transaction field formats	22
TYPE	10, 46, 74, 75, 76, 77, 84
Type blank segments	55
Type S0 segments	54
Typing ahead	35
U	
Unique segments	47, 48, 49, 50, 59, 93
UPDATE	44, 45
Updating segment instances	45
V	
VALIDATE	10, 46, 59, 61, 63, 64, 65, 67, 68, 69, 70, 73, 74
Validation tests	91
Variables.....	113
W	
WITH-UNIQUES method	49, 50, 59