TIBCO FOCUS®

Maintaining Databases

Release 8207.27.0
March 2021
DN1001059.0321

Contents

Chapter 1

Modifying Data Sources With MODIFY

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

The MODIFY facility is also used to maintain data in relational structures, Adabas data sources, and VSAM data sources. See documentation for specific data adapters for details about using MODIFY in those environments.

MODIFY can also be used to load fixed format sequential data sources that consist of a single segment. Data is loaded in the order in which it is input. Update and delete operations are not supported with this type of data source. If the file already exists, new data is loaded at the end. In order to append data to a sequential data source with HiperFOCUS ON, the record format must be fixed.

In this chapter:

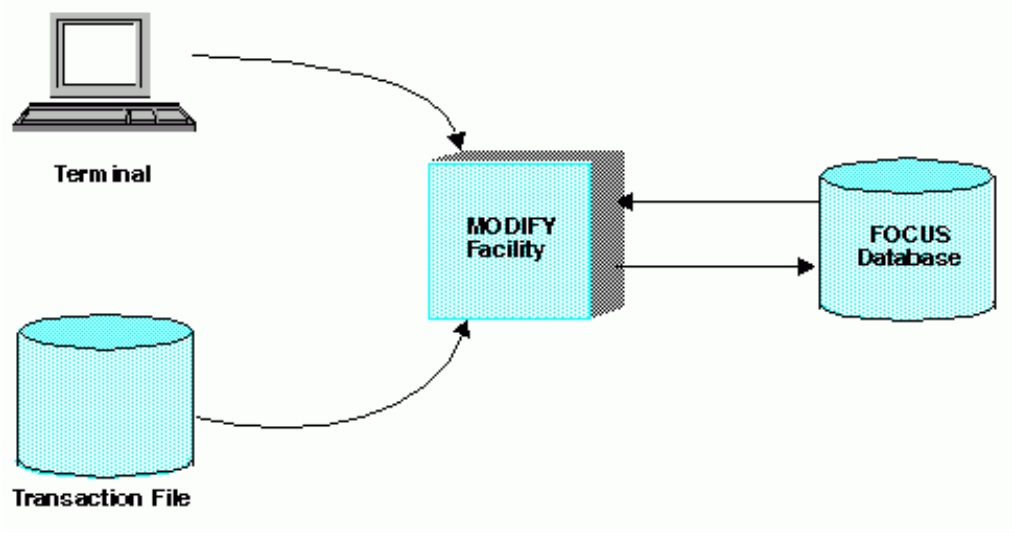
Introduction

A MODIFY request processes a transaction in three steps:

1. It reads a transaction for incoming data values. Transactions can come from external data sources, may be supplied by the user in screens or in response to prompts, or can be included as part of the request itself.

2. It selects a segment instance for changing or deleting, or confirms that a segment instance does not exist yet in the data source.
3. It changes or deletes the segment instance it selected, or adds a new segment instance.

This is shown graphically in the following diagram:



The request first reads a transaction (that is, a related collection of incoming data values). *Describing Incoming Data* on page 33 describes the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements that describe transactions read by the request.

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

- It searches the data source for segment instances containing the same values as the transaction. This is done with a MATCH statement.
- It selects the next segment instance after the current position. This is done with a NEXT statement.

The MATCH and NEXT statements are discussed in *Modifying Data: MATCH and NEXT* on page 75.

The request then either adds, updates, or deletes data source values using the incoming values, or it rejects the transaction.

Examples of MODIFY Processing

This section provides examples of MODIFY processing that add, update and delete data from a data source.

Each request indicates the data source it is modifying, the method of reading data, the transaction values it searches for in the data source, and the actions it takes depending on whether the values are in the data source or not. If it is reading a transaction data source, the request must indicate the name of the data source.

Adding Data to a Data Source

The following sample MODIFY request adds new employee data to the EMPLOYEE data source. When you run the request, it prompts you for an employee ID number, last name, and first name. After you enter these three values, the request adds the information to the data source and prompts you for three more values for the same fields. When you are finished entering data, end execution by entering the word END to any prompt.

The request is as follows:

```
1. MODIFY FILE EMPLOYEE
2. PROMPT EMP_ID LAST_NAME FIRST_NAME
3. MATCH EMP_ID
4.    ON MATCH REJECT
5.    ON NOMATCH INCLUDE
6. DATA
```

The parts of the request are as follows:

1. The MODIFY FILE EMPLOYEE statement indicates that the request modifies the EMPLOYEE data source.
2. The PROMPT statement indicates that the request will prompt you for the employee's ID (EMP_ID), last name, and first name on the terminal.
3. The MATCH EMP_ID statement searches the data source for the employee ID that you entered.
4. If the ID is already in the data source (that is, an ID in the data source matches the ID you entered), the MATCH statement rejects your transaction.
5. If the ID is not yet in the data source, the MATCH statement adds your transaction to the data source.
6. The DATA statement begins prompting for data.

Updating Data in a Data Source

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

- The EMP_ID field contains the employee ID number. It is the first nine characters on the record.
- The DEPARTMENT field contains the new department assignment, and is the next ten characters.
- The CURR_SAL field contains the new salary, and is the last eight characters.

This is the EMPDEPT data source:

```
* * * TOP OF FILE * * *
071382660PRODUCTION27500.00
112847612SALES     24800.75
451123478MARKETING 26950.00
* * * END OF FILE * * *
```

The request is as follows:

```
    MODIFY FILE EMPLOYEE
1.  FIXFORM EMP_ID/9 DEPARTMENT/10 CURR_SAL/8
2.  MATCH EMP_ID
2.     ON NOMATCH REJECT
2.       ON MATCH UPDATE DEPARTMENT CURR_SAL
3.  DATA ON EMPDEPT
4.  END
```

The parts of the request are as follows:

1. The FIXFORM statement indicates that the transaction records are in fixed positions in the EMPDEPT data source and describes the positions of the fields in each record.
2. The MATCH EMP_ID statement searches the data source for the employee ID in each record. If the ID is not in the data source, the request rejects the record. If the ID is in the data source, the request replaces the DEPARTMENT and CURR_SAL values in the data source with the values on the record.
3. The DATA statement indicates that the data is contained in the data source EMPDEPT. EMPDEPT is the ddname to which the data file is allocated, and can be different from the system file name.
4. The END statement completes the request and initiates processing.

Deleting Data From a Data Source

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

```
    MODIFY FILE EMPLOYEE
1.  PROMPT EMP_ID
2.  MATCH EMP_ID
       ON MATCH DELETE
       ON NOMATCH REJECT
3.  DATA
```

The parts of the request are as follows:

1. The PROMPT statement indicates that the request will prompt you for the employee's ID.
2. The MATCH statement searches for the employee ID in the data source. If the ID is in the data source, the request deletes all information relating to the employee from the data source.
3. The DATA statement begins prompting for data.

Additional MODIFY Facilities

You can also instruct the request to perform other tasks:

- Test transaction values to determine whether they are acceptable. You do this using the VALIDATE statement, described in *Computations: COMPUTE and VALIDATE* on page 106.
- Perform calculations and store the results in either transaction or temporary fields. You do this using the COMPUTE statement, described in *Computations: COMPUTE and VALIDATE* on page 106.
- Display messages that contain values from transaction fields, temporary fields, or data source fields. You do this using the TYPE statement, discussed in *Messages: TYPE, LOG, and HELPMESSAGE* on page 130.
- Record transactions processed by the request using the TYPE and LOG statements described in *Messages: TYPE, LOG, and HELPMESSAGE* on page 130. These statements can sort accepted transactions from rejected transactions and can sort rejected transactions by reason for rejection.

You can design MODIFY requests using Case Logic, a method which divides requests into sections called "cases." The request can branch to the beginning of a case during execution. Case Logic, discussed in *Case Logic* on page 145, makes it possible for requests to offer the terminal operator selections and to process transactions in different ways.

You can design MODIFY requests that process multiple segment instances at one time. Multiple Record Processing is described in *Multiple Record Processing* on page 169, including the modification of several segment instances on one FIDEL screen.

Reference: **Notes on Using JOIN Syntax With MODIFY**

For software that supports the MODIFY facility, note the following:

- The JOIN command allows you to read (but not to modify) data in a second FOCUS data source using the MODIFY LOOKUP function. To modify multiple FOCUS data sources in one request, use the COMBINE command.
- The LOOKUP function in MODIFY requests cannot be used on a DEFINE-based JOIN; DEFINE is not evaluated during a MODIFY procedure.
- The MODIFY LOOKUP function cannot retrieve data in a cross-referenced segment using concatenated fields (a multi-field join).

FOCUS offers a variety of other advanced features that facilitate use of the MODIFY command in more complex applications. These features are listed below and described in *Advanced Facilities* on page 195:

- The COMBINE command for modifying multiple FOCUS data sources in one MODIFY request.
- The COMPILE command for translating MODIFY requests into compiled code ready for execution.
- The ACTIVATE and DEACTIVATE statements for activating and deactivating fields.
- The Checkpoint and Absolute File Integrity facilities and the COMMIT and ROLLBACK Subcommands for protecting FOCUS data sources from system failures.
- The ECHO facility for displaying the logical structure of MODIFY requests.
- Dialogue Manager system variables that record execution statistics every time a MODIFY request is run.
- FOCUS query commands that display statistical information on MODIFY request executions and FOCUS data sources.

The rest of this introduction contains:

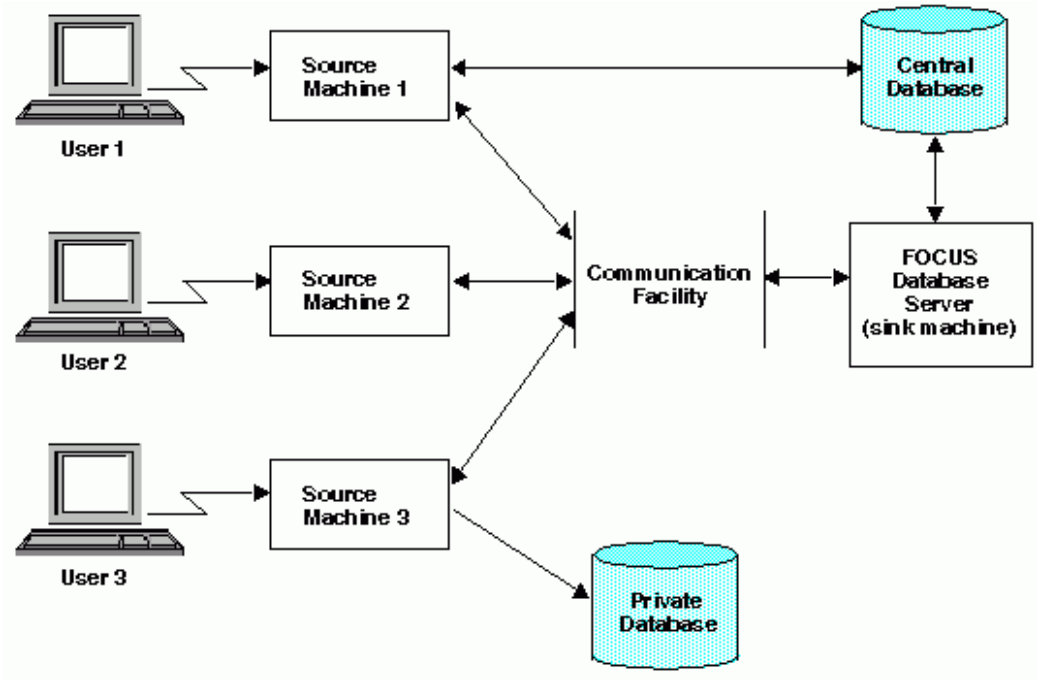
- The basic syntax of MODIFY requests.
- Instructions for executing MODIFY requests.

- A summary of facilities other than MODIFY that can be used to maintain FOCUS data sources.
- A short description of the parts of the EMPLOYEE data source most used in the examples.

Multiple User Access

Suppose you need to update a particular data source, but three other users have been assigned to work on the data source at the same time. How can you be sure that one user's changes will not override or overwrite another user's changes? MODIFY, used in conjunction with the Simultaneous Usage (SU) facility, ensures data integrity under those circumstances.

To enter SU mode, you initiate a background job process called a FOCUS Database Server. The user ids running FOCUS or Host Language Interface programs are called source machines. The users (using their source machines) send requests and transactions to the FOCUS Database Server, which processes the transactions and transmits the retrieved data or messages back to the source machine. The following diagram illustrates the process:



Under SU, when you run a MODIFY request

1. The request identifies the instance to be changed with MATCH or NEXT commands.

2. The source machine forwards the transaction values to the FOCUS Database Server, which uses the values to retrieve the correct instance.
3. The FOCUS Database Server retrieves the original data source instance, holds one copy, and sends another to the source (user id) that requested the data.
4. The source machine updates its copy of the instance with the new field values, or marks the copy for deletion and sends the updated copy back to the FOCUS Database Server. The FOCUS Database Server compares the copy of the instance that it saved with the instance stored in the data source to check whether the data source instance has since been updated by another user.

   At this point, two courses of action are possible:

   - If the copy and the current instance in the data source are the same, FOCUS changes the instance using the copy from the source machine.
   - If the original and the current instance in the data source are different, SU signals a conflict and rejects the source machine copy.

Notice that a source machine may work on separate, locally controlled data sources.

Reference: SU Features

With SU you can display a list of the active source machine userids and the fields of the FOCUS Database Server data sources from your source machine, and record all user actions in a sequential data source called HLIPRINT. The HLIPRINT data source records each user action, the data source on which the action took place, the segment read or modified by the action, and the user id that issued the action. It can also include the:

- Date and time of the action.
- CPU time it took to execute the action.
- Number of I/O operations required to execute the action.
- Name of the FOCUS stored procedure executing the action, and the name of the case executing the action (for MODIFY requests using Case Logic).

Another SU feature is the FOCURRENT variable that alerts users to transaction conflicts. When you submit a MODIFY transaction in SU, FOCUS stores a value in a variable called FOCURRENT to indicate what happened to the transaction. You can design your MODIFY requests to test FOCURRENT and take different actions, depending on whether the transaction was accepted or rejected. The following request tests the FOCURRENT variable:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL
CASE NEWSAL
MATCH EMP_ID
    ON NOMATCH REJECT
    ON MATCH PROMPT CURR_SAL
    ON MATCH UPDATE CURR_SAL
    ON MATCH IF FOCURRENT EQ 0 GOTO TOP;
    ON MATCH TYPE
      "VALUE CHANGED. NEW VALUE <D.CURR_SAL>"
ENDCASE
DATA
```

The request prompts for an employee ID and then branches to the case NEWSAL. If the ID is in the data source, you are prompted for the current salary of the employee; the current salary is updated on the source machine copy. The transaction is submitted.

Next, the request tests the values of the variable FOCURRENT:

- If FOCURRENT is 0, the transaction is accepted and the request prompts you for the next EMP_ID.
- If FOCURRENT is not 0, the transaction is rejected. The request branches back to the top of the procedure. If the instance is found, FOCUS prompts for the current salary and resubmits the transaction. If the instance was deleted, the request reports back a NOMATCH condition and prompts you for the next transaction.

By testing the FOCURRENT variable, MODIFY requests can process transactions after they have been rejected because of conflicts.

Managing Your Data: Advanced Features

In addition to the basic operations of the MODIFY facility, many other features are available to help you refine your MODIFY requests. This section describes them briefly.

Feature	Description
Absolute File Integrity	Causes FOCUS to write changes to the data source to another section of the disk rather than overwriting the data source. If the request executes normally, the new section of the disk becomes part of the data source. If the system fails, the original data source is preserved.
ACTIVATE	Activates an inactive transaction field. It declares a transaction field to be present so the transaction field can be used for matching, including, and updating. The MOVE option equates the transaction value of the transaction field to the corresponding data source field. The RETAIN option does not move the data source value to the transaction field.
DEACTIVATE (RETAIN)	Deactivates a transaction field. The DEACTIVATE command changes a transaction value to blank if alphanumeric, to zero if numeric, or to the MISSING transaction value for transaction fields described by the MISSING=ON attribute. It also deactivates the corresponding data source field. The RETAIN option deactivates the field without changing its value.
CHECK	Limits the number of transactions lost if the system fails when you are modifying a data source by identifying a checkpoint. CHECK activates the Checkpoint facility that enables FOCUS to write more frequently to the data source. (The point at which the transactions are written is called the "checkpoint.") The Checkpoint Facility is useful in cases when a system failure occurs while MODIFY requests are executing.
COMBINE	Enables you to modify multiple FOCUS, relational, or Adabas data sources in one MODIFY request.

Feature	Description
COMMIT and ROLLBACK	Control the changes made to data sources and protect the data sources from system failures. COMMIT and ROLLBACK improve SU performance; here the ability to group individual transactions as one logical transaction reduces the number of individual transactions and the amount of communication needed between the FOCUS Database Server and source userids. COMMIT and ROLLBACK are used in lieu of CHECK.
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, relational, or Adabas data sources and makes these values available for other computations.
ECHO	Displays the logical structure of MODIFY requests. This feature is a good debugging tool for analyzing a MODIFY request, especially if the logic is complex and MATCH and NEXT defaults are used.
FIND	Searches another FOCUS, relational, or Adabas data source for the presence of the transaction value.
LOG	Enables you to record transactions and error messages in separate files automatically, and to control the display of rejection messages at the terminal.
MULTIPLE RECORD PROCESSING COMMANDS	Enable you to process multiple segment instances at one time and are often used with CRTFORM. A few of the important commands used in multiple record processing are GETHOLD and REPEAT. GETHOLD retrieves transaction records from memory and uses them to modify a data source; GETHOLD collects and retrieves segment instances. REPEAT does re-iterative processing.
TYPE	Displays or stores messages in a separate file that you prepare.

Feature	Description
VALIDATE	Enables you to reject transactions that contain unacceptable values.

MODIFY Command Syntax

The general syntax of the MODIFY command is

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

where:

`MODIFY FILE`

Begins the request.

`filename`

Is the name of the FOCUS, relational, fixed format sequential, VSAM, or Adabas data source you are modifying. This name must be the same as the Master File of the data source. For information about modifying non-FOCUS data sources, see the appropriate data adapter documentation.

Note: Although you can use MODIFY to load a fixed format sequential file, the sequential data source must consist of a single segment, and data is loaded in the order in which it is input. Update and delete operations are not supported. To append data to an existing sequential data source with HiperFOCUS ON, the record format must be fixed.

`ECHO`

Invokes the ECHO facility, which displays the request logic (see *Displaying MODIFY Request Logic: The ECHO Facility* on page 213).

`TRACE`

Invokes the TRACE facility, which displays the name of each case that is entered during the execution of the request if the request uses Case Logic (see *Tracing Case Logic: The TRACE Facility* on page 167).

statements

Are the MODIFY statements in the request. Each statement must begin on a separate line.

DATA

Specifies the source of incoming data. Note that nothing should come between this statement and the END statement, unless you are supplying the incoming data in the request itself. In that case, place the data after the DATA statement.

ON *ddname*

Is a DATA statement parameter. See *Specifying the Source of Data: The DATA Statement* on page 72.

VIA *program*

Is a DATA statement parameter.

incoming data

Is the data you are using to modify the data source if you are supplying the data in the request itself.

END

Concludes the request. Do not add this statement if the request contains PROMPT statements (PROMPT statements are discussed in *Prompting for Data One Field at a Time: The PROMPT Statement* on page 58).

Executing MODIFY Requests

You can enter and run a MODIFY request either by entering it at the terminal or by running it as a stored procedure (stored procedures are discussed in the *Developing Applications* manual). When you start execution of the request, FOCUS executes the request for each transaction until:

- There is no more data to be read in the incoming transaction data source (the file containing the incoming data).
- The user signals a halt (if the request is prompting the user for data).
- The STOP statement signals a halt to the processing of transactions in an incoming data source (see *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* on page 73).
- The request encounters a GOTO EXIT statement.

Syntax: How to Execute a Request as a Stored Procedure

To enter a MODIFY request as a stored procedure, type the request in a procedure file (procedures are discussed in the *Developing Applications* manual). If you are including the incoming data in the request (which you might do for testing purposes), place the data after the DATA statement in the stored procedure. End the request with the END statement unless the request contains PROMPT statements.

After saving the file, enter at the FOCUS prompt

```
EX focexec
```

where *focexec* is the name of the stored procedure.

FOCUS responds with an echo of the file name, date, and time as follows:

```
filename ON date AT time
```

The request then either begins prompting you for data or starts reading the stored transactions.

When the request finishes execution, it displays the following statistics

```
TRANSACTIONS:  TOTAL  = n ACCEPTED  = n REJECTED = n
SEGMENTS: INPUT       = n UPDATED   = n DELETED  = n
```

where:

n

Is an integer.

TRANSACTIONS

Are the transactions processed by the request.

TOTAL

Is the total number of transactions processed.

ACCEPTED

Is the number of transactions accepted by the request and used to maintain the data source.

REJECTED

Is the number of transactions rejected by the request.

SEGMENTS

Is the number of segment instances modified by the request.

`INPUT`

Is the number of new segment instances.

`UPDATED`

Is the number of instances updated.

`DELETED`

Is the number of instances deleted.

To suppress this message, include the following command in the procedure before the MODIFY request:

```
SET MESSAGE = OFF
```

Syntax: How to Execute MODIFY Requests Online

To execute a MODIFY request online, enter

```
MODIFY FILE filename
```

where

filename

is the FOCUS name of the data source you are modifying.

FOCUS responds with an echo of the data source name, date, and time as follows:

```
filename ON date AT time
ENTER SUBCOMMANDS:
```

Enter each MODIFY statement in the request (such as FIXFORM, MATCH, COMPUTE, TYPE) followed by a DATA statement and the incoming data (if the data is not coming from another data source or from the terminal). Then enter the END statement (unless the request contains PROMPT statements).

The request can then start prompting you for data, read from an external data source, or accept transaction records from the terminal (if the request contains FIXFORM or FREEFORM statements but does not specify the ddname of an external data source).

If it accepts transaction records from the terminal, the request appears:

```
START:
```

Start entering the data, one record at a time. Every time you enter a record, the request processes it and displays a message if it rejects the record. After you have entered the data, enter the END statement. This ends execution.

If you are entering a MODIFY request online and you want to cancel the request and start over, enter QUIT. This returns you to the FOCUS prompt.

If you enter a statement online that FOCUS considers an error, it will prompt you for a correction. This error correction facility is described in the *Creating Reports* manual.

You should not enter MODIFY requests online unless the requests are short. If you enter a statement you want to change, you must quit the request and start over.

The example below shows a sample MODIFY request being entered online:

```
>
modify file employee

 EMPLOYEEFOCUS A1 ON 08/15/85 AT 16.36.05
 ENTER SUBCOMMANDS:
freeform emp_id curr_sal
match emp_id
on nomatch reject
on match update curr_sal
data
 START:
emp_id=071382660, curr_sal=21400.50, $
emp_id=112847612, curr_sal=20350.00, $
emp_id=117593129, curr_sal=22600.34, $
end
```

Notice that when the request finishes execution, it displays the following statistics:

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

These statistics are explained in the preceding section.

Other Ways of Maintaining FOCUS Data Sources

Although the MODIFY command is one of the primary methods of maintaining FOCUS data sources, there are four other facilities for changing data in FOCUS data sources:

- The Maintain facility allows you to maintain data sources (including FOCUS, DB2, SQL/DS, Oracle, Teradata, and VSAM data sources) using event-driven and set-based processing in with a Graphical User Interface. The Maintain facility is described in *Introduction to Maintain*, through Expressions Reference.
- The FSCAN and SCAN facility allows you to edit FOCUS data sources interactively on a field-by-field basis. You enter a subcommand to make each change. The facility can update key fields. The FSCAN facility is the subject of *Directly Editing FOCUS Databases With FSCAN* on page 389. SCAN is the subject of *Directly Editing FOCUS Databases With SCAN* on page 349.

- The Host Language Interface (HLI) allows you to maintain FOCUS data sources from computer programs written in BAL, FORTRAN, COBOL, and PL/1. HLI is covered in the *Host Language Interface Users Manual*.

Unlike the FSCAN facility mentioned above, the MODIFY command allows you to make many changes with one execution. It can run in both interactive and batch modes. It will prompt you for the values it needs to make the changes, or it may read the values from a transaction data source. However, it cannot update key fields.

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

You can only MODIFY one partition of a partitioned FOCUS data source at one time. You must explicitly allocate the partition to be modified. Alternatively, you can create separate Master Files for each partition for use in MODIFY procedures. For more information about partitioned FOCUS data sources, see the *Describing Data* manual.

The EMPLOYEE Data Source

The examples in this chapter use the EMPLOYEE data source, a data source used to record employee information for a company. The Master File and the diagram of the entire data source structure are shown in Master Files and Diagrams. Most of the examples use three segments in the EMPLOYEE data source:

- The EMPINFO segment contains information directly relating to employees in a company: employee ID, last name, first name, hire date, department assignment, current salary, job code, and classroom hours.
- The SALINFO segment contains information relating to employees' monthly pay: the pay date and the amount of pay.
- The DEDUCT segment contains information about the deductions taken off each monthly pay check: the type of deduction and the amount of the deduction.

Describing Incoming Data

This section describes the statements that read and describe transactions. These are the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements. The last part of the section discusses the DATA, START, and STOP statements.

To modify a data source, the MODIFY request first reads incoming data. It then uses this data to select the segment instances that must be changed or deleted, or to confirm that the instances have not been entered yet and to add them. The data may be in fixed or comma-delimited format, it may be stored in sequential data sources or within the request itself, and it may be entered directly by users on terminals.

There are four MODIFY statements that read and describe incoming data. Some read data from sequential data sources and the request itself; some prompt users on terminals for data. They are:

`FIXFORM`	Reads data in fixed format. That is, the fields occupy fixed positions in each record.
`FREEFORM`	Reads data in comma-delimited format. That is, the fields in each record are separated by a comma (,). Each record is terminated by a comma and a dollar sign (,$).
`PROMPT`	Prompts users on terminals for data values one field at a time. This statement works on all terminals.
`CRTFORM`	Displays formatted screens (called CRTFORMs) on terminals and allows users to enter multiple data values at one time.

Note: PROMPT, FREEFORM, FIXFORM, and CRTFORM statements accept data that includes numbers expressed in scientific notation. For more information on the use of scientific notation in expressions, refer to the *Creating Reports* manual.

If a request does not have one of these statements, it defaults to FREEFORM and reads data from a comma-delimited list.

These statements can be placed in requests in two ways:

- The statements can stand by themselves. These statements read data every time the request repeats.
- The statements can be phrases in MATCH or NEXT statements (discussed in *Modifying Data: MATCH and NEXT* on page 75). These phrases only read data when the MATCH or NEXT statement is executed.

A request may have an unlimited number of statements of one type (for example, 10 PROMPT statements), except for CRTFORM where up to 255 such statements are allowed. You may also mix the following statements in one request:

- FREEFORM statements and PROMPT statements.
- One FIXFORM statement with up to 255 CRTFORMs.

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

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

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

The last part of the section discusses several other features related to reading transactions. They are:

- The DATA statement that marks the end of the executable portion of the request and specifies the source of the transactions (the request itself, a data source, the terminal, or a user program).
- The START and STOP statements that limit the request to reading a portion of the transaction data source.

Reading Fixed-Format Data: The FIXFORM Statement

The FIXFORM statement reads data in fixed format. That is, each field has a fixed position in each record. The FIXFORM statement can read data from sequential data sources, including HOLD, SAVE, and SAVB files generated by TABLE requests.

The FIXFORM statement reads in one logical record at a time starting from column one and divides the record into transaction fields. Subsequent FIXFORM statements may redefine the record, dividing it into different sets of fields.

Note: Multiple FIXFORM statements in a request can function as a single statement.

For example, you are adding the names of five new employees to the EMPLOYEE data source. The data is stored in a sequential data source called NEWEMP.

This is how the data source appears on a text editor such as TED:

```
|....+....1....+....2....+....3....+....4
* * * TOP OF FILE * * *
222333444BLACK      SUSAN     27500.00
456456456NEWMAN     JERRY     24800.75
999888777HUNTINGTON LAWRENCE  26950.00
246246246LINDQUIST  DEBRA     19300.40
666888222MCINTYRE   GEORGE    31900.60
* * * END OF FILE * * *
```

Each record in the data source consists of four fields, each field in a fixed position on the record:

- The EMP_ID field (employee ID numbers) occupies the first nine bytes of each record (columns 1 through 9).
- The LAST_NAME field occupies the next ten bytes (columns 10 through 19).
- The FIRST_NAME field occupies the next ten bytes (columns 20 through 29).
- The CURR_SAL field (current salaries) occupies the last eight bytes in each record (columns 30 through 37).

You can describe the record format with this FIXFORM statement:

```
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/10 CURR_SAL/8
```

To add the records to the FOCUS data source, include the preceding statement in this MODIFY request:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 LAST_NAME/10 FIRST_NAME/10 CURR_SAL/8

MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA ON NEWEMP
END
```

Syntax: How to Use a FIXFORM Statement

The syntax of the FIXFORM statement is

```
FIXFORM [ON ddname] fld-1/form-1 ... fld-n/form-n
```

or

```
FIXFORM FROM master [ALIAS]
```

where:

fld-1 ...

Are the names of the incoming data fields that the FIXFORM statement is reading or redefining. If the name has an embedded blank, enclose it within single quotation marks.

Any field being read by the FIXFORM statement that does not appear in the Master File of the data source being modified must be predefined in a COMPUTE field/format=; statement. This COMPUTE must appear in the MODIFY before the FIXFORM.

The list of fields must fit on one line. If the list is too long to fit on one line, use a FIXFORM statement for each line. For example:

```
FIXFORM EMP_ID/9 LAST_NAME/15
FIXFORM CURR_SAL/8 ED_HRS/4
```

The two FIXFORM statements act as one statement and read one record into the buffer.

form-1 ...

Are the formats of the incoming data fields, as described in *How to Specify Field Formats With FIXFORM* on page 41. The formats specify the format type (alphanumeric, integer, floating point, and so on) and the length of the field in bytes.

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

To specify an alphanumeric format, type the length of the field in bytes. For example, a record contains two alphanumeric fields:

The EMP_ID field, nine bytes long.

The DEPARTMENT field, ten bytes long.

The FIXFORM statement that describes this record is:

```
FIXFORM EMP_ID/9 DEPARTMENT/10
```

Note that alphanumeric transaction fields can modify any data source field regardless of internal format. Specifying the formats of binary, packed, and zoned transaction fields is discussed in *How to Specify Field Formats With FIXFORM* on page 41.

Remember that a transaction field can contain numbers and still be alphanumeric. If you display a transaction data source on a system editor, alphanumeric data appears normally; numeric data appears as unprintable hexadecimal characters.

ON *ddname*

Is an option that specifies the ddname of the transaction data source containing the incoming data. You use this option most often when the request is reading data from two different sources: one source is specified by the DATA statement, the other by the ON ddname option.

Note that if there is more than one FIXFORM statement without the ON ddname option, the request keeps track of the last column of the physical record read by the last FIXFORM statement. If the last column is in the middle of the record, the next FIXFORM statement begins to read from the next column. If the last column is at the end of the record, the next FIXFORM statement begins to read from column 1 of the next record.

To break a FIXFORM statement having the ON ddname option into smaller statements, specify the ON ddname option only in the first statement. All the statements must be together in one block. For example:

```
FIXFORM ON EMPFILE EMP_ID/9 LAST_NAME/15
FIXFORM FIRST_NAME/10 DEPARTMENT/10
FIXFORM CURR_SAL/8 ED_HRS/4
```

FROM *master*

Indicates that the incoming data fields have the same names and formats as the Master File (named *master*). If you use this option, do not specify the field names and formats in the FIXFORM statement itself. Use this option only if the Master File specifies a single segment SUFFIX=FIX data source. All the fields in the Master File specified by the FROM phrase must also appear in the Master File specified by the MODIFY command, or an error will result.

You use this option most often to load data from a HOLD file. For example:

```
TABLE FILE EMPLOYEE
PRINT CURR_SAL BY EMP_ID
ON TABLE HOLD
END
MODIFY FILE SALARY
FIXFORM FROM HOLD
DATA ON HOLD
END
```

The TABLE request stores employee IDs and salaries in a HOLD file. The MODIFY request loads the IDs and salaries into a new FOCUS data source called SALARY. Note that all the fields in the HOLD Master File must also appear in the SALARY Master File.

Text fields are supported with FIXFORM from HOLD; only one text field can be read from a HOLD file and it must be the last field on the HOLD FIXFORM. The representation of missing text depends on whether MISSING=ON in the Master File or the FIXFORM format is C for conditional, or a combination of the two.

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

Note: FIXFORM FROM Master File automatically assumes that all fields on the FIXFORM are conditional fields. Because of this a value of blank does not update the database to a value of blank. If blank (or spaces) is a valid value, and the update should take place, you must issue an ACTIVATE RETAIN fieldname fieldname fieldname... or ACTIVATE RETAIN SEG.fieldname.

`ALIAS`

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

Note: If the transaction file has a null (missing data) value for a file, and you want to input this value as a blank, the Master Files for both the transaction file and the data source being modified must have MISSING=ON for that field.

Syntax: How to Skip Columns in the Record

Often, an incoming transaction contains filler or data you do not need. To skip over characters or information in the incoming record, type

X*n*

where:

n

Is the number of columns you want to skip.

This does not cause the statement to ignore the skipped columns. The statement reads the entire record; it just does not place the skipped data in any transaction field. Later in the request, you can place this data into transaction fields by adding a second FIXFORM statement (see the following section, *How to Move Backward Through a Record* on page 40).

For example, a transaction record consists of two fields: EMP_ID and CURR_SAL. Two "A"s separate the fields:

```
071382660AA23540.35
```

You describe this record with this FIXFORM statement:

```
FIXFORM EMP_ID/9 X2 CURR_SAL/8
```

The X2 notation prevents the two "A"s from being placed in the transaction fields.

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

Procedure: How to Move Backward Through a Record

After a FIXFORM statement reads a record into the buffer, it places the data into transaction fields, starting from the beginning of the record and moving toward the end. You can specify that FIXFORM back up a number of columns to process the data more than once. This enables you to place the same data into two fields simultaneously. To do this, use the notation

```
X-n
```

where *n* is the number of columns that the statement is to move backward. For example, the first three digits of employee IDs are a special code that you wish to use later in the request. Each employee ID is nine digits long. You type this FIXFORM statement:

```
FIXFORM EMP_ID/9 X-9 EMP_CODE/3 X6 CURR_SAL/8
```

A record in the transaction data source is:

```
07138266023500.35
```

The statement interprets the record this way:

`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, you must define the field with the COMPUTE statement before the FIXFORM statement (see *Computing Values: The COMPUTE Statement* on page 106).

You may replace any FIXFORM statement with two smaller statements so that the second statement redefines all or part of the record read by the first statement. For example, you may replace this FIXFORM statement

```
FIXFORM EMP_ID/9 X-9 EMP_CODE/3 X6 CURR_SAL/8
```

with these two smaller FIXFORM statements:

```
FIXFORM EMP_ID/9 CURR_SAL/8
FIXFORM X-17 EMP_CODE/3 X14
```

The first FIXFORM statement reads one record and divides the record into the EMP_ID field (nine bytes) and the CURR_SAL field (eight bytes).

The second FIXFORM statement moves 17 bytes back to the beginning of the record and declares the first three bytes to be the EMP_CODE field. It then skips over the last 14 bytes.

Note that you cannot place the X-*n* notation at the end of a FIXFORM statement. The following statement is an error:

```
FIXFORM EMP_ID/9 CURR_SAL/8 X-17
```

FIXFORM statements that redefine records in the buffer are especially useful in Case Logic requests (see *Case Logic Applications* on page 159).

Syntax: How to Specify Field Formats With FIXFORM

This section lists the data formats that may be specified in FIXFORM statements. In addition to alphanumeric format, there are date (DATE), text field (TX), and conditional text field (CTX) formats, and numeric formats of fields in HOLD and SAVB files and of fields generated by user-written programs. The formats are

```
[A]n[YQMDWV] In[YQMD] F4 D8 Pn[.m][YQMD] DATE /TX /CTX Zn[.m]
```

where:

`[A]n[YQMD]`

Specifies an alphanumeric character string *n* bytes long, where *n* is an integer.

Date component options (YY, Y, Q, M, D) are included as necessary for a date field.

The V and W options are for A*n*V fields that were propagated to a HOLD file.

- W indicates that the length of the input field is *n*+6 bytes. The first six bytes contain the length of the character data within the subsequent *n* bytes. Use for inputting data from HOLD FORMAT ALPHA files.
- V indicates that the length of the input is *n*+2 bytes. The first two bytes are binary and contain the length of the character data within the subsequent *n* bytes. Use for inputting data from binary HOLD 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 for a date field.

`F4`

Specifies a 4-byte binary floating point number.

`D8`

Specifies an 8-byte binary double precision number.

`Pn[.m][YQMD]`

Specifies a packed number *n* bytes long with *m* digits after an implied decimal point. *n* is an integer between 1 and 16 and *m* is an integer between 0 and 33. Date component options (YY, Y, Q, M, D) are included as necessary for a date field.

`DATE`

Specifies a date field in 4-byte integer format, to be copied to the data source without date translation or validation. Date format fields can also be read without these restrictions by specifying alphanumeric, integer, or packed format, as described later in this section.

`/TX|/CX`

Specifies a text field format for transaction and conditional transaction fields. Each FIXFORM statement can include multiple text fields. However, they must appear as the last fields in the statement, they *may not* be conditional, and, in the data file, each text field must be terminated with the %$ character combination on a line by itself. Note that you do not specify the length when using FIXFORM to read text fields; the length is for display purposes only (see the *Describing Data* manual).

See *Entering Text Data Using TED* on page 69 for general rules.

Note:

- Text fields must be the last fields listed in the FIXFORM statement. If they are being loaded from a HOLD file, they must also be the last fields in the HOLD file.
- If the word END appears on a line by itself, FOCUS interprets it as a quit action, stops the procedure, and discards everything entered up to that point for a particular record.
- To end a transaction and exit MODIFY, first enter the end-of-text character (%$) on a line by itself, then enter END on the next line.
- If data is read from an external data source, the record format must be fixed.

- If a text field is not mentioned in the FIXFORM statement, but it is present in the Master File, the value of the text field is determined based on the setting of the MISSING attribute. That is, if MISSING=ON, the text will be entered as a dot (.). If MISSING=OFF, the text will be entered as a blank.

`Zn[.m]`

Specifies a zoned decimal number *n* bytes long with *m* digits after an implied decimal point. *n* is an integer between 1 and 16 and *m* is an integer between 0 and 9.

For example, this FIXFORM statement

```
FIXFORM EMP_ID/9 HIRE_DATE/I4 CURR_SAL/D8 ED_HRS/P4.2
```

defines each record as the following:

- The first nine bytes as the character string EMP_ID.
- The next four bytes as the binary integer HIRE_DATE.
- The next eight bytes as the binary double precision number CURR_SAL.
- The next four bytes as the packed number ED_HRS. The last two digits of the number follow an implied decimal point.

The FIXFORM statement specifies the field formats of transaction data sources, not the data source being updated. A transaction field can modify a data source field if the transaction field has one of the following format types (the format type is the type of field, such as alphanumeric or floating point):

- The same format type as the data source field.
- Alphanumeric format.
- Zoned format (if the data source field is packed).

If you specify any other format type for the transaction field (for example, an integer transaction field to modify a floating point data source field), the request may terminate and generate an error message. To read such a transaction value into a data source field, do the following:

1. Before the FIXFORM statement, use the COMPUTE statement to define a name for the incoming data field that is different from the data source field (the COMPUTE statement is discussed in *Computations: COMPUTE and VALIDATE* on page 106). The statement also specifies the field format, showing the format type and the number of digits in the field.
2. In the FIXFORM statement, read the incoming data field using the name you defined in the COMPUTE statement. The field format in the FIXFORM statement shows the field length in bytes in the transaction data source.

3. After the FIXFORM statement, use the COMPUTE statement to set a field with the same name as the data source field equal to the value of the field you defined in step 1.

   Note: If the incoming field is numeric and the data source field is alphanumeric, use the EDIT function to do this. The EDIT function is described in the *Creating Reports* manual.

The following request reads a floating point field called FLOATSAL into the data source double-precision field CURR_SAL:

```
MODIFY FILE EMPLOYEE
COMPUTE FLOATSAL/F8=;
FIXFORM EMP_ID/12 FLOATSAL/F4
COMPUTE CURR_SAL = FLOATSAL;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA ON FLOAFILE
END
```

Notice that the FLOATSAL field is defined with a format of F8 in the first COMPUTE statement and a format of F4 in the FIXFORM statement. FLOATSAL is an eight-digit field that takes up four bytes in the transaction data source.

Controlling Whether FIXFORM Input Fields Are Conditional

In MODIFY, by default, FIXFORM FROM *mastername* treats all transaction data as conditional, meaning that space-filled fields are considered not present, and as such cannot be updated or used in updates.

The SET FIXFRMINPUT command enables you to specify how to handle FIXFORM input fields as either conditional (field/format C) or non-conditional fields. Thus, spaces in a transaction field can be used for updating database fields.

Syntax: How to Control Whether FIXFORM Input Fields Are Conditional

```
SET FIXFRMINPUT = {COND|NONCOND}
```

where:

`COND`

Treats all transaction fields generated by FIXFORM FROM *mastername* as conditional (format C) fields. COND is the default value.

`NONCOND`

Treats all transaction fields as present in the transaction, and their contents are treated as real values.

Note that if you have not changed the value of the FIXFRMINPUT parameter and you query its value, the value displays as DEFAULT.

Reference: **Usage Notes for SET FIXFRMINPUT**

- The FIXFRMINPUT setting does not affect a FIXFORM command that does not have a FROM phrase.
- If you run a compiled MODIFY, its behavior reflects the FIXFRMINPUT setting at the time it was compiled, even if a different setting is in effect at run time.

Example: **Controlling Whether FIXFORM Transaction Fields Are Conditional**

The following procedure establishes a transaction file, defining LN1 in HOLD file TRANS to be blank for PIN 000000040.

```
SET ASNAMES = ON
DEFINE FILE EMPDATA
LN1/A15 = IF PIN EQ '000000040' THEN '' ELSE LN;
END
TABLE FILE EMPDATA
PRINT PIN LN1 AS LN
IF PIN FROM '000000010' TO '000000100'
ON TABLE HOLD AS TRANS
END
```

The following procedure, sets the FIXFORM FROM input fields as conditional (the default) and reports on the output from the MODIFY:

```
SET FIXFRMINPUT = COND
-? SET FIXFRMINPUT &FIXF

MODIFY FILE EMPDATA
 FIXFORM FROM TRANS
 MATCH PIN
  ON MATCH UPDATE LN
  ON NOMATCH REJECT
 DATA ON TRANS
END



TABLE FILE EMPDATA
HEADING
" "
"VALUE OF FIXFRMINPUT IS &FIXF "
" "
PRINT PIN LN
 IF PIN FROM '000000010' TO '000000100'
END
```

The output shows that the blank in the transaction file was not used to update the last name in the data source:

```
 VALUE OF FIXFRMINPUT IS COND

PIN        LASTNAME
---        --------
000000010  VALINO
000000020  BELLA
000000030  CASSANOVA
000000040  ADAMS
000000050  ADDAMS
000000060  PATEL
000000070  SANCHEZ
000000080  SO
000000090  PULASKI
000000100  ANDERSON
```

The following procedure sets the FIXFORM FROM input fields as non-conditional and reports on the output from the MODIFY:

```
SET FIXFRMINPUT = NONCOND
-? SET FIXFRMINPUT &FIXF

MODIFY FILE EMPDATA
 FIXFORM FROM TRANS
 MATCH PIN
  ON MATCH UPDATE LN
  ON NOMATCH REJECT
 DATA ON TRANS
END

TABLE FILE EMPDATA
HEADING
" "
"VALUE OF FIXFRMINPUT IS &FIXF "
" "
PRINT PIN LN
 IF PIN FROM '000000010' TO '000000100'
END
```

The output shows that the last name for PIN 000000040 has been updated to contain blanks:

```
 VALUE OF FIXFRMINPUT IS NONCOND

PIN        LASTNAME
---        --------
000000010  VALINO
000000020  BELLA
000000030  CASSANOVA
000000040
000000050  ADDAMS
000000060  PATEL
000000070  SANCHEZ
000000080  SO
000000090  PULASKI
000000100  ANDERSON
```

Describing Date Fields

This section discusses using date format fields in FIXFORM statements. Alphanumeric and integer format fields with date edit options are not discussed here; they are treated by FIXFORM like standard alphanumeric and integer fields.

When you use a FIXFORM statement to modify a data source date field, the corresponding data in the transaction data source can be one of the following three types:

- A numeric date literal. For example, August 17 1989 can be represented in the transaction data source as 081789. The transaction field format can be A*n*, I*n*, or P*n*.
- A natural date literal. For example, August 17 1989 can be represented in the transaction data source as AUG 17 1989. The transaction field format must be A*n*.

  Note that all names of days and months in the transaction data source must be in uppercase, even if the translation option is *t* or *tr*. All abbreviated names of days and months in the transaction data source must consist of the first three letters of the name. Commas cannot be included in the date.
- A date in internal FOCUS date format. This format is used for date fields in SAVB and unformatted HOLD files. The date is stored as a 4-byte integer representing the elapsed time since the standard FOCUS base date, as described in the *Describing Data* manual. The transaction field format must be DATE.

  For example, assume that you have changed the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FIXFORM statement:

  ```
  FIXFORM EMP_ID/11 FIRST_NAME/10 LAST_NAME/10 HIRE_DATE/9
  ```

  Both of these records are valid input:

```
444555666 DOROTHY TAILOR 860613
444555666 DOROTHY TAILOR 86 JUN 13
```

To describe date fields in FIXFORM statements, you can use the following transaction field formats.

- DATE. This specifies a transaction field stored in FOCUS internal date format, which is a 4-byte integer representing the time elapsed from the standard FOCUS base date, as described in the *Describing Data* manual. The transaction field will be copied directly to the data source without date validation.

  For example:

  ```
  FIXFORM SALEDATE/DATE
  ```

- A*n*, I*n*, P*n*. These specify a date field stored in alphanumeric, integer, or packed decimal format respectively. Numeric date literals and natural date literals are translated as necessary to suit the data source field's USAGE specification and edit options.

  For example, if a data source contains the date field NEWSDATE, and USAGE=MDYY, the following FIXFORM statements can be used to update NEWSDATE:

  ```
  FIXFORM NEWSDATE/A8YYMD
  FIXFORM NEWSDATE/A6DMY
  FIXFORM NEWSDATE/I4MDY
  FIXFORM NEWSDATE/I2YMD
  FIXFORM NEWSDATE/P3DMY
  FIXFORM NEWSDATE/A8
  ```

  Note that the last FIXFORM statement does not specify any date components. Because it is alphanumeric and has the same length specified by the data source field's USAGE attribute, it defaults to the USAGE format (which in this case is MDYY).

For all date transaction field formats, the date components (year, quarter, month, day) do not need to be in the order specified in the USAGE attribute in the Master File; they can be in any order.

Note, however, that you cannot extract date components from a date field (for example, you cannot write a YMD transaction field to a YM data source field), and you cannot convert one component to another (for example, you cannot convert a YM transaction field to a YQ data source field). The only exceptions are the YY and Y date components, which can be substituted for each other.

Syntax: How to Describe Repeating Groups

You may use a fixed-format transaction record to modify multiple segment instances. The set of transaction fields that modify the instances is called a repeating group because the fields repeat for each instance. Instead of explicitly specifying each field, you specify the repeating group once with a multiplying factor in front.

The syntax is

```
FIXFORM factor (group)
```

where:

factor

Is the number of times that the group repeats.

group

Is the repeating group consisting of a list of fields and formats.

For example, assume you design a request that records the last 12 months of employees' monthly pay in the EMPLOYEE data source. Each transaction record contains the employee's ID and 12 pairs of fields: the first field in each pair is the pay date, the second is the monthly pay (GROSS). The request is:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 12 (PAY_DATE/6 GROSS/7)
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA ON EMPGROSS
END
```

Each incoming record that the request reads contains one EMP_ID field and 12 groups of fields, each group consisting of a pay date field and a monthly pay field. The request reads a record, then splits the record into 12 smaller logical records, each consisting of the employee ID of the original record and one group. FOCUS then executes the request for each logical record, processing each group separately.

You may specify more than one group in a FIXFORM statement, but they cannot be nested.

Note: To process repeating groups in a Case Logic request, place each repeating group in a FIXFORM statement in a separate case. The case should include the following:

- A counter that counts the group being processed.

- An IF statement that branches out of the case after all the groups are processed.
- GOTO phrases that branch back to the beginning of the case after each group is processed.

The following request adds and updates information on employees' monthly pay. Note the ON INVALID phrase that branches back to the beginning of the case if a monthly pay entry is greater than $2500. The request is:

```
MODIFY FILE EMPLOYEE
COMPUTE
  COUNTER/I3 = 0;
FIXFORM EMP_ID/9
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO NEWPAY
GOTO NEWPAY

CASE NEWPAY
COMPUTE
  COUNTER/I1 = COUNTER + 1;
IF COUNTER GT 3 GOTO TOP;
FIXFORM 3 (PAY_DATE/6 GROSS/7)
VALIDATE
  PAYTEST = IF GROSS GT 2500 THEN 0 ELSE 1;
  ON INVALID GOTO NEWPAY
MATCH PAY_DATE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO NEWPAY
  ON MATCH UPDATE GROSS
  ON MATCH GOTO NEWPAY
ENDCASE
DATA ON PAYFILE
END
```

Using Date Format Fields

The following examples show how to use date format fields.

Example: Conditional Fields

MODIFY requests can process records in which alphanumeric field values may be present in one input record but absent in another. Such fields are called conditional fields. When the value of a conditional field is blank, the request does not use the field to modify the data source and the field remains inactive (active and inactive fields are discussed in *Active and Inactive Fields* on page 204).

To indicate to FOCUS that a field is conditional, precede the field format with the letter C. For example:

```
FIXFORM FIRST_NAME/C10 LAST_NAME/C15
```

Another example: You design a MODIFY request that updates employees' departments and job codes. If an employee's department or job code has not changed, the corresponding field in the transaction data source is blank.

The request is:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 DEPARTMENT/C10 X1 CURR_JOBCODE/C3
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_JOBCODE
DATA
071382660 SALES   B13
112847612 A08
117593129 MARKETING
END
```

The request contains three incoming records after the DATA statement:

- The first incoming record contains all three fields. The request updates both the DEPARTMENT and CURR_JOBCODE fields.
- The next record has the EMP_ID and CURR_JOBCODE fields but no DEPARTMENT field. The request updates the employee's CURR_JOBCODE value in the data source, but leaves the DEPARTMENT value the same.
- The last record has the EMP_ID and DEPARTMENT fields but no CURR_JOBCODE field. The request updates the employee's DEPARTMENT value in the data source, but leaves the CURR_JOBCODE value the same.

If you did not describe the DEPARTMENT and CURR_JOBCODE fields as conditional, the request would change an employee's department or job code to blank whenever these fields in the incoming records were blank.

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

- Blank, if the instance fields are alphanumeric.
- Zero, if the instance fields are numeric.
- The MISSING symbol, if the fields are described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

Example: FIXFORM Phrases in MATCH and NEXT Statements

You may use FIXFORM statements as phrases in MATCH and NEXT statements. These phrases are useful if you want to read records selectively only if a particular segment instance exists in the data source (or is confirmed not to be in the data source).

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

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 PAY_DATE/6
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE

MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH FIXFORM ON MONTHPAY GROSS/7
  ON NOMATCH INCLUDE

DATA ON EMPPAY
END
```

The data is kept in two transaction data sources: EMPPAY and MONTHPAY. The EMPPAY data source contains the employee IDs and the date each employee was paid. The MONTHPAY data source contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

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

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

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

and record it in the data source:

```
ON NOMATCH INCLUDE
```

Reading in Comma-delimited Data: The FREEFORM Statement

The FREEFORM statement reads comma-delimited data, where field values in each record are separated by commas, and records are terminated by comma-dollar signs (,$). The data may be stored in the request itself or in separate sequential data sources.

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

The following request updates employee salaries by reading employee IDs and new salaries from comma-delimited records. The records follow the DATA statement:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
EMP_ID=071382660, CURR_SAL=21400.50, $
EMP_ID=112847612, CURR_SAL=20350.00, $
EMP_ID=117593129, CURR_SAL=22600.34, $
END
```

Syntax: How to Use a FREEFORM Statement

The syntax of the FREEFORM statement is

```
FREEFORM [ON ddname] [field-1field-2 ... field-n]
```

where:

`ON ddname`

Is an option that specifies the ddname of the transaction data source containing the incoming data. Use this option only when the DATA statement does not specify a ddname or specifies a ddname of a different data source.

field-1 ...

Are the names of the fields in the order that they appear in the record.

Note: FREEFORM follows the same rules as FIXFORM when dealing with TEXT fields. For more information see *Reading Fixed-Format Data: The FIXFORM Statement* on page 35.

If the order of fields is specified in the data, you do not need it in the syntax and if the order of fields is specified in the syntax, you do not need it in the data.

The list of fields must fit on one line. If the list is too long for a single line, use a FREEFORM statement for each line. For example:

```
FREEFORM EMP_ID LAST_NAME FIRST_NAME
FREEFORM DEPARTMENT CURR_SAL
```

These two FREEFORM statements act as one statement and read one record into the buffer.

Each time a FREEFORM statement is executed, it reads one record up to the comma-dollar sign (,$). It does not read beyond that. If the FREEFORM command is used with incoming data having embedded commas, the data must be enclosed in single quotation marks in the input data source.

If a MODIFY request has a FREEFORM statement, the statement must specify all the fields in the transaction data source. If the transaction data source has fields not specified in the FREEFORM statement, the request terminates and generates an error message.

If you do not include a transaction statement in your MODIFY request, the request assumes the default FREEFORM and expects to read comma-delimited data. The request reads one record every time it executes the first statement in the request. Nevertheless, you should include a FREEFORM statement to make clear that the request is reading comma-delimited data, to show when the request reads the data, and to allow greater flexibility in entering data into comma-delimited data sources.

If the Master File lists a date format with a translation option (see the *Describing Data* manual), you can type the date values in the transaction data source as they appear in reports generated by TABLE requests (but do not type the commas in the dates). Note the following conditions:

- The date format must have had the translation option before the FOCUS data source was created.
- All names of months must be in uppercase, even if the translation option is *t* or *tr*.

For example, assume you change the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FREEFORM statement:

```
FREEFORM EMP_ID FIRST_NAME LAST_NAME HIRE_DATE/9
```

Both these records are valid input:

```
444555666, DOROTHY, TAILOR, 860613, $
444555666, DOROTHY, TAILOR, 86 JUN 13, $
```

Identifying Values in a Comma-delimited Data Source

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

- Identified values are identified explicitly in the data source.

- Positional values exist by themselves without any identification.

Identified values have the form

```
identifier = value
```

where *identifier* identifies the field to which the value belongs.

Identifiers can be one of two types:

- Field names or unique truncations of field names. For example:

```
DEPARTMENT=SALES, CURR_SAL=25000, $
```

- Aliases. For example:

```
DPT=SALES, CSAL=25000, $
```

If the request has a FREEFORM statement, the statement must specify all identified fields. However, the request identifies the values by their identifiers, not by the order of field names in the FREEFORM list.

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

```
SALES, 25000, $
```

The MODIFY request identifies positional values by the order of field names specified in the FREEFORM statement list. If a record consists only of positional values, the request assigns the first field name in the list to the first value, the second field name in the list to the second value, and so on. For example, if a request has the statement:

```
FREEFORM EMP_ID DEPARTMENT CURR_SAL
```

Then the record

```
071382660, SALES, 25000, $
```

is interpreted this way:

```
EMP_ID: 071382660
DEPARTMENT:  SALES
CURR_SAL:   25000
```

If a record has both identified and positional values, the MODIFY request identifies the positional values in the following way: it notes the last explicitly identified value to precede the positional values in the record. It then identifies the positional values by the order of field names that follow the name of the explicitly identified field in the FREEFORM list.

For example, a MODIFY request has this FREEFORM statement:

```
FREEFORM EMP_ID FIRST_NAME LAST_NAME CURR_SAL
```

The transaction data source contains this record:

```
FIRST_NAME=DAVID, MCHENRY, 21300.45, $
```

The first value, DAVID, is explicitly identified as the FIRST_NAME field. The request identifies the next value, MCHENRY, as the LAST_NAME field because LAST_NAME follows FIRST_NAME on the FREEFORM list. Similarly, the request identifies 21300.45 as the CURR_SAL field. The EMP_ID field retains the value it was last given.

If the MODIFY request has no FREEFORM statement, it identifies positional values by the order of field names declared in the Master File. If a record consists of only positional values, the request assigns the first field name in the Master File to the first value, the second field name to the second value, and so on. For example, a transaction data source contains this record:

```
071382660, MCHENRY, DAVID, $
```

The request identifies the first value, 071382660, as the EMP_ID field because EMP_ID is the first field in the Master File. The next value, MCHENRY, is the LAST_NAME field (the second field in the Master File). DAVID becomes the FIRST_NAME field, the third field in the Master File (the EMPLOYEE Master File is shown in Master Files and Diagrams).

If a record has both identified values and positional values, the MODIFY request identifies the positional values the following way: it notes the last explicitly identified value to precede the positional values in the record. It then identifies the positional values by the order of field names that follow the name of the explicitly identified field in the Master File. For example, the transaction data source contains this record:

```
FIRST_NAME=DAVID, 820406, PRODUCTION, $
```

The first value, DAVID, is explicitly identified as the FIRST_NAME field. The request identifies the next value, 820406, as the HIRE_DATE field because HIRE_DATE follows FIRST_NAME in the Master File. Similarly, the request identifies PRODUCTION as the DEPARTMENT field.

Example: Missing Values in Comma-delimited Data Sources

If a field value is missing for a particular record, you must explicitly identify the name of the next field in the record. For instance, a FREEFORM statement specifies the following:

```
FREEFORM EMP_ID CURR_SAL DEPARTMENT
```

One record lacks a CURR_SAL value. Type the record this way

```
071382660, DEPARTMENT=PRODUCTION, $
```

where 071382660 is an EMP_ID value. The CURR_SAL field remains inactive and will not change any CURR_SAL values in the data source.

If you are adding segment instances to the data source, the instance fields not receiving a value become:

- Blank, if the instance fields are alphanumeric.
- Zero, if the instance fields are numeric.
- The MISSING symbol, if the fields are described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

An important exception: If you omit fields from the beginning of a record, the fields retain the values last assigned to them from a previous record. For example, a transaction data source contains these two records:

```
EMP_ID=071382660, PAY_DATE=820831, GROSS=1045.60, $
PAY_DATE=820831, GROSS=1047.20, $
```

The second record is lacking an EMP_ID value. Nevertheless, since EMP_ID is at the beginning of the record, it retains its value of 071382660 for the second record and remains active.

If you use double commas to mark an absent value, the value becomes a blank character string if alphanumeric, and zero if numeric. Note that the request can use this value to modify the data source. For example, in the record

```
071382660,, PRODUCTION, $
```

the two commas mark the position of the absent CURR_SAL field. The CURR_SAL field becomes active and can change an employee salary to $0.00.

Example: FREEFORM Phrases in MATCH and NEXT Statements

You may use FREEFORM statements as phrases in MATCH and NEXT statements. These phrases are useful if you want to read records selectively if a particular segment instance exists in the data source (or is confirmed not to be in the data source).

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

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID PAY_DATE
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH FREEFORM ON MONTHPAY GROSS
  ON NOMATCH INCLUDE
DATA ON EMPPAY
END
```

The data is kept in two transaction data sources: EMPPAY and MONTHPAY. The EMPPAY data source contains the employee IDs and the date each employee was paid. The MONTHPAY data source contains the amount each employee was paid (GROSS). The request must confirm for every EMPPAY transaction that:

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

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

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

and record it in the data source:

```
ON NOMATCH INCLUDE
```

Prompting for Data One Field at a Time: The PROMPT Statement

The PROMPT statement prompts the user on a terminal for incoming data one field at a time. Use this statement for requests that may be run on line terminals or by users having no access to the FIDEL facility. If the requests will be run exclusively by users on full-screen terminals with access to FIDEL, use the CRTFORM statement instead. The FIDEL facility and the CRTFORM statement are the subjects of *Designing Screens With FIDEL* on page 227.

Syntax: How to Use a PROMPT Statement

The syntax of the PROMPT statement is

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

where:

field-1 ...

Are the names of the fields for which you are prompting. An asterisk * instead of field names prompts for all fields described in the Master File in the order that they are declared.

The list of fields must fit on one line. If the list is too long to fit on one line, use a PROMPT statement for each line. For example:

```
PROMPT EMP_ID LAST_NAME FIRST_NAME
PROMPT DEPARTMENT CURR_SAL
```

Each field in the Master File with a text field format must appear in a separate PROMPT statement as the last field in the statement. When prompted for text, note that the length of the text entry is limited only by the amount of virtual storage space. The last line of text data that you enter must be followed by the end-of-text mark (%$) on a line by itself. For additional guidelines regarding fields with a text field format, see *Entering Text Data Using TED* on page 69.

text

Is optional prompting text, up to 38 characters per field.

Do not place an END statement at the end of the request. Conclude the request with the DATA statement.

The following request updates information about employees' department assignments, salaries, and job codes:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT CURR_SAL CURR_JOBCODE
MATCH EMP_ID
  ON MATCH UPDATE DEPARTMENT CURR_SAL CURR_JOBCODE
  ON NOMATCH REJECT
DATA
```

When you execute the command, the following appears on your screen

```
> EMPLOYEE ON 06/19/98 AT 14.38.27
DATA FOR TRANSACTION 1

MP_ID= >
```

where:

EMPLOYEE

Is the system name of the data source (in this case, the TSO name).

```
ON 06/19/98 AT 14.38.27
```

Is the date and time that FOCUS opened the data source: June 19, 1998 at 2:38:27 p.m.

```
DATA FOR TRANSACTION 1
```

Notifies the user that the request is prompting for the first transaction. Each cycle of prompts constitutes one transaction. When the next transaction begins, the request prompts again for the first field in the cycle. In this request, the EMP_ID, DEPARTMENT, CURR_SAL, and CURR_JOBCODE prompts constitute one transaction. When the next transaction begins, the request prompts for the EMP_ID field again.

```
EMP_ID = >
```

Is the default prompt for the EMP_ID field (the field name).

As each prompt appears, enter the value for the field requested. When you finish entering values, end execution by entering End or Quit at any prompt. The following is a sample execution of the request shown above (user input is shown in lowercase; computer responses are in uppercase):

```
> EMPLOYEE ON 06/19/98 AT 14.38.27
    DATA FOR TRANSACTION 1

    EMP_ID        = > 071382660
    DEPARTMENT    = > mis
    CURR_SAL      = > 22500.35
    CURR_JOBCODE  = > b12
    DATA FOR TRANSACTION 2

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

When you design a request that prompts for fields and validates them, we recommend that validating the field values after every prompt is recommended. This saves extra typing if one of the field values proves invalid. Validation tests are discussed in *Validating Transaction Values: The VALIDATE Statement* on page 114.

If the Master File lists a date format with a translation option (see the *Describing Data* manual), you may type the date as it appears in reports generated by TABLE requests (but do not type the commas in the dates). Note that the date format must have had the translation option before the FOCUS data source was created.

For example, assume you change the format of the HIRE_DATE field in the EMPLOYEE Master File from I6YMD to YMDT. You then write a request that creates a new EMPLOYEE data source. The request begins with this FIXFORM statement:

```
PROMPT EMP_ID FIRST_NAME LAST_NAME HIRE_DATE
```

When you execute the request, a sample transaction might appear like this:

```
DATA FOR TRANSACTION 2

EMP_ID           = > 444555666
FIRST_NAME       = > dorothy
LAST_NAME        = > tailor
HIRE_DATE (YMDT) = > 98 jun 13
```

Note that you can also respond to the HIRE_DATE prompt with the value 980613.

Syntax:

How to Prompt for Repeating Groups

You may prompt for the same group of fields repeatedly. This is convenient when you want to modify a child segment chain. You prompt once for the key field of the parent instance and prompt repeatedly for the values of the child instances. Without repeating groups, you must prompt for the key field of the parent instance each time you prompt for a child instance.

For example, a MODIFY request updates employees' monthly pay. It first prompts for an employee ID, then for 12 pairs of fields: the first field in each pair is a pay date, the second field is the updated pay. The pay date and updated pay fields are a repeating group.

To specify a repeating group, use the following syntax

```
PROMPT factor (group)
```

where:

factor

Is the number of times the group repeats.

group

Is the repeating group of fields.

Note that the transaction counter that appears during prompting counts each repeating group cycle of prompts as one transaction.

For example, the following request adds three instances of monthly pay (GROSS) for each employee:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID 3 (PAY_DATE GROSS)
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

This request prompts you for an employee ID, then a pay date, a monthly pay, a pay date, a monthly pay, and so on until it prompts you for three pay dates and three monthly pays. It then prompts you for the next employee ID.

The following is a sample execution of the previous request:

```
> EMPLOYEE  ON 09/19/98 AT 15.01.38
    DATA FOR TRANSACTION   1

    EMP_ID      = > 071382660
    PAY_DATE    = > 860131
    GROSS       = > 1360.50
    DATA FOR TRANSACTION   2

    PAY_DATE    = > 860228
    GROSS       = > 1360.85
    DATA FOR TRANSACTION   3

    PAY_DATE    = > 860331
    GROSS       = > 1360.50
    DATA FOR TRANSACTION   4

    EMP_ID      = >
```

You can place multiple repeating groups in the same statement. This PROMPT statement contains two repeating groups:

```
PROMPT EMP_ID 3 (PAY_DATE GROSS) 2 (DAT_INC SALARY)
```

The statement prompts for:

1. An employee ID.
2. A pay date and a monthly pay, three times.
3. A salary raise date (DAT_INC) and a new salary, two times.
4. The next employee ID.

You can nest repeating groups. For example, this prompt statement

```
PROMPT EMP_ID 6 (PAY_DATE 7 (DED_CODE DED_AMT))
```

prompts for:

1. An employee ID.
2. A pay date.
3. A deduction code and deduction amount, seven times.
4. Steps 2 and 3 repeat for a total of six times.
5. The next employee ID.

Syntax:

How to Prompt Text

When you run a request containing PROMPT statements, the request prompts you for each field by displaying the field name and an equal sign (=). However, you may specify your own prompt. The syntax is

```
PROMPT fieldname.text.
```

where:

fieldname

Is the name of the field you are prompting for.

text

Is the text you want to appear as the prompt, up to 38 characters. Text must be enclosed within periods.

Note the following rules regarding prompt text:

- The text must be delimited by a period (.) on either side, with no space between the field name and the first period.
- The text cannot contain apostrophes or single quotation marks (').
- The text must be typed on one line.
- A single MODIFY request can contain up to 4000 characters of prompt text.

This request adds new employees to the EMPLOYEE data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID.ENTER THE EMPLOYEE ID NUMBER:.
PROMPT FIRST_NAME.ENTER FIRST NAME:.
PROMPT LAST_NAME.ENTER LAST NAME:.
PROMPT HIRE_DATE.WHAT DATE WAS EMPLOYEE HIRED?.
PROMPT CURR_SAL.WHAT IS THE STARTING SALARY?.


MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Special Responses

This section discusses special responses to prompts. It covers:

- Canceling a transaction

- Ending execution
- Correcting a field value
- Typing ahead
- Repeating the last response
- Entering no data
- Breaking out of repeating groups
- Invoking the FIDEL Facility

Canceling a Transaction

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

```
(FOC309) TRANSACTION INCOMPLETE:
```

and will prompt you for the next transaction. Canceling a transaction clears the buffer of data and causes the PROMPT statement to re-prompt you for the fields, allowing you to clear a bad transaction and start over.

Ending Execution

To end execution of the request, enter either Quit or End after any prompt. The request displays the execution statistics and returns you to the FOCUS command level. The data source will be updated to the last completed transaction.

Correcting Field Values

If you entered an incorrect field value, you can correct it at the next prompt. Type the value for the next prompt, but do not press *Enter*. Instead, type a comma and then type

```
fieldname = corrected-value
```

where:

fieldname

Is the field name of the corrected value. Then press *Enter*. Note that *fieldname* must be separated from the previous value by a comma.

The example below shows a user correcting a DEPARTMENT value after the CURR_JOBCODE prompt.

```
> DATA FOR TRANSACTION  1

    EMP_ID        = > 071382660
    DEPARTMENT    = > production
    CURR_SAL      = > 19350.67
    CURR_JOBCODE  = > a03, department=sales
    DATA FOR TRANSACTION  2

    EMP_ID        = >
```

Note: If you enter an incorrect field value at the last prompt of a transaction, you cannot correct the value in that transaction.

Typing Ahead

You can enter several values at one prompt by typing ahead. *Enter*

value-1, value-2, ... value-n

where:

value-1

Is the value of the field for which you are being prompted.

value-2 ...

Are the values of fields you have not yet been prompted for by the PROMPT statement. The values must be in the order of fields specified by the PROMPT statement, from the field being prompted for onwards. Separate the values with commas.

For example, a MODIFY request has this PROMPT statement:

```
PROMPT EMP_ID DEPARTMENT CURR_SAL CURR_JOBCODE
```

When you run the request, you enter an employee ID, a department, salary, and job code at the EMP_ID prompt, as shown below.

```
> DATA FOR TRANSACTION 1

    EMP_ID      = > 071382660, sales, 23800, b04
    DATA FOR TRANSACTION 2

    EMP_ID      = >
```

Repeating a Previous Response

If you are going to respond to a prompt with the same value as the previous prompt, you may enter a double quotation mark (") instead to save typing.

Entering No Data

If you run a request that prompts you for a field that should not contain data, enter a period (.) after the prompt. The field becomes inactive and does not change any values in the data source.

If you are adding segments to the data source, the field in the new instance becomes:

- Blank, if the instance field is alphanumeric.
- Zero, if the instance field is numeric.
- The MISSING symbol, if the field is described with the MISSING=ON attribute in the Master File (see the *Describing Data* manual).

Breaking Out of Repeating Groups

To break out of a repeating group, enter an exclamation point (!) after any prompt. The request will immediately prompt you for the first field outside the repeating group.

For example, you run this request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID 3 (PAY_DATE GROSS)
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Every time you enter an employee ID, the request prompts you for a pay date and a monthly pay (GROSS) three times. If you enter an exclamation point at one of these prompts, the request prompts you for the next employee ID.

Each cycle of prompts within a repeating group counts as one transaction. The repeating group data you entered before the transaction where you broke out remains active and modifies the data source.

If you break out of one repeating group nested in another repeating group, the request next prompts you for the fields of the outer group. For example, a request contains this PROMPT statement:

```
PROMPT EMP_ID 6 (PAY_DATE 7 (DED_CODE DED_AMT))
```

You run the request. If you enter an exclamation point at a DED_CODE or DED_AMT prompt, the request next prompts you for the next PAY_DATE value.

Reference: PROMPT Phrases in MATCH and NEXT Statements

You can use PROMPT statements as phrases in MATCH or NEXT statements. By doing so, you avoid prompting the user for data that will be rejected anyway. The following examples illustrate the differences.

Consider the following request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL

MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

This request prompts the user for the EMP_ID and CURR_SAL fields. The MATCH statement searches the data source for the EMP_ID value the user enters (MATCH EMP_ID). If it finds the value, it updates the CURR_SAL value; otherwise it rejects the transaction. The user must enter both an EMP_ID and a CURR_SAL value every transaction, whether the transaction is accepted or not.

However, when the request prompts for the CURR_SAL value in the MATCH statement, the user enters a CURR_SAL value only if the corresponding EMP_ID value is in the data source. This request shows how this is done:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID


MATCH EMP_ID
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

The request prompts you for an EMP_ID value. It then searches the data source for the ID you entered. If it does not find the value, it rejects the ID and prompts you for another ID. Only if it finds the ID in the data source does it prompt you for a CURR_SAL value.

Reference: Using PROMPT and FREEFORM Statements in One Request

You may use PROMPT and FREEFORM statements together in one request. This feature is useful when key field values are difficult to read and type, such as large numbers or complex codes. For example, a request might read employee ID numbers from a comma-delimited data source, use those IDs to locate segment instances, and then prompt the user for the data to update the employee information.

To use FREEFORM and PROMPT together, follow these rules:

- Place all FREEFORM statements before the PROMPT statements.
- Place the data in a separate data source. Specify the data source with the ON ddname option.
- Do not end the comma-delimited records with dollar signs ($).

Note that when you use FREEFORM together with PROMPT, the transaction counter does not appear before the prompts.

This request updates employee salaries:

```
MODIFY FILE EMPLOYEE
FREEFORM ON EMPNO EMP_ID

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE "ENTER SALARY FOR EMPLOYEE #<EMP_ID"
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
DATA
```

Note the TYPE phrase in the MATCH statement that informs the user what employee ID the request is processing. The TYPE statement is described in *Displaying Specific Messages: The TYPE Statement* on page 131.

Invoking the FIDEL Facility: The CRTFORM Statement

This section is a brief description of the CRTFORM statement, which is discussed fully in *Designing Screens With FIDEL* on page 227.

The CRTFORM statement invokes the FIDEL facility, which generates a formatted screen. You type the transaction values in the designated areas of the screen and press *Enter*.

To use the FIDEL facility, you must be on a full-screen terminal running FOCUS in interactive mode, not batch. Note that FIDEL is separate from the MODIFY facility, so your installation may have MODIFY but not FIDEL. Consult your systems manager or database administrator.

Beneath the CRTFORM statement, you specify the layout of the screen. Enclose each line of the screen in double quotation marks. On each line, you can type free text instructing the user and designate data entry areas where the user enters data for specific fields.

You may also display messages to the user in the TYPE area of the CRTFORM using the HELPMESSAGE attribute (see *Displaying Messages: Setting PF Keys to HELP* on page 145 and in the *Describing Data* manual).

The following request updates employees' department assignments, salaries, job codes, and classroom hours:

```
MODIFY FILE EMPLOYEE
CRTFORM
" ***** EMPLOYEE INFORMATION UPDATE *****"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"
"ENTER EMPLOYEE'S DEPARTMENT: <DEPARTMENT"
"ENTER CURRENT SALARY: <CURR_SAL"
"ENTER JOB CODE: <CURR_JOBCODE"
"ENTER CLASS HOURS: <ED_HRS"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON MATCH UPDATE CURR_JOBCODE ED_HRS
DATA VIA FI3270
END
```

A request may have up to 255 CRTFORM statements, and may also have one FIXFORM statement preceding the CRTFORM statements. You may place CRTFORM phrases in MATCH and NEXT statements.

The FIDEL facility has several features that enhance its usability:

- Turnaround fields display field values as they exist in the data source, which you can then change.
- Display fields display field values that you cannot change. You can use these fields to design CRTFORM screens for data source inquiry.
- Screen attributes display different parts of the screen in different colors, highlighted, underlined, or flashing.
- Multiple-record processing allows you to modify several segment instances on one screen.

Please refer to *Designing Screens With FIDEL* on page 227, to learn how to use FIDEL.

Entering Text Data Using TED

While in MODIFY, TED can be used to enter text field data. When TED is used to enter text, a new temporary file is opened in memory for data input; this file is never written to disk permanently. The name of this file is the same as the name of the text field. The ddname for the text field will be TXTFLD. For example

```
DESCRPT TXTFLD
```

is the file name and file type of the file opened for the text field DESCRIPT.

All TED rules and functions apply, including the ability to edit other files. The RUN function in TED is ignored for text fields and is treated as the FILE command instead.

There are six ways to use the syntax for entering text format data using TED:

```
TED textfield
ON MATCH TED textfield
ON NOMATCH TED textfield
ON MATCH/NOMATCH TED textfield
ON NEXT TED textfield
ON NONEXT TED textfield
```

For example:

```
MODIFY FILE COURSES
PROMPT COURSE_CODE
MATCH COURSE_CODE
  ON NOMATCH TED DESCRIPTION
  ON NOMATCH INCLUDE
  ON MATCH TED DESCRIPTION
  ON MATCH UPDATE DESCRIPTION
DATA
```

TED will always edit the most recent version of the text field. The first time, this will be the current data source text field value; the next time that TED is used on the same text field, data from the previous text transaction will be available for editing.

As a rule, TED will always look for text data in the transaction area first. If no text exists there, TED looks for text present as a result of MATCH. If there is no data there, TED assumes that the field is new and brings up a new (empty) file.

After one transaction involving TED is complete, data areas are blanked out before proceeding with the next transactions (as when DEACTIVATE is used). This means that all text instances will be newly created (therefore, one course description will not carry over and accidentally be used for the next course number).

Text fields must always end with the end-of-text mark (%$). Although you may enter this mark directly in the TED file as the first two characters on the last line, TED will test for the presence of the end-of-text mark; if it is missing, TED automatically inserts it.

Note: You must supply the end-of-text mark when using PROMPT or FIXFORM.

If you wish to use TED to input data for more than one text field, specify a separate action for each field:

```
ON MATCH TED TXFIELD1
ON MATCH TED TXFIELD2
```

The size of the file is limited only by the amount of available storage space.

Entering Text Field Data

The following rules apply to text field data entry using TED, FIXFORM, FREEFORM, or PROMPT:

- You can begin entering text data at any position on a line.
- Leading blanks on a line are preserved.

  A line will be treated as the start of a new paragraph if it starts with three or more blanks. To prevent the concatenation of lines when a text field is displayed, insert at least three blanks at the beginning of each line.
- Blank lines are permitted.

Defining a Text Field

The syntax for defining a text field in a Master File is:

```
FIELD=fieldname, ALIAS=aliasname, FORMAT=TXnn,$
```

or

```
FIELD=fieldname, ALIAS=aliasname,FORMAT=TXnnF,$
```

where:

fieldname

Is the name you assign the text field.

aliasname

Is an alternate name for the field name.

nn

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

F

Is used to format the text field for redisplay when TED is called using ON MATCH or ON NOMATCH. When F is specified, the text field is formatted as TX80 and is displayed. When F is not specified, the field is redisplayed exactly as entered.

Displaying Text Fields

FOCUS includes a format option in the text field of the Master File. Use of this determines whether text will display in the format in which it was entered.

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

```
FILE=TEXT,SUFFIX=FOC
   SEGNAME=SEGA,SEGTYPE=S1
   FIELD=KEYFLD,,A1,$
   FIELD=TXTFLD,,TX20,$
```

Sample data entered:

```
THIS IS A TEST OF THE NEW TED OPTION 'F'.  REMEMBER THAT TED DISPLAYS 80
CHARACTERS ON THE SCREEN.  THREE LEADING BLANKS ARE USED TO INDICATE A
NEW PARAGRAPH. TEXT FIELD DATA IS ALWAYS STORED EXACTLY AS ENTERED.  WHEN
F IS INCLUDED IN THE FORMAT AND THE TEXT FIELD IS REDISPLAYED, BLANKS ARE
OMITTED AND THE FIELD IS CONDENSED.
WHEN F IS NOT INCLUDED, THE FIELD IS REDISPLAYED AS ENTERED.
```

Since the text field in the Master File does not include the F option, the data will be redisplayed exactly as entered using TED (ON MATCH TED TXTFLD).

For the next example, the text field includes the F option:

```
FILE=TEXT,SUFFIX=FOC
   SEGNAME=SEGA,SEGTYPE=S1
   FIELD=KEYFLD,,A1,$
   FIELD=TXTFLD,,TX20F,$
```

Note: The same data is entered as in the previous example.

In this case, since the text field does include the F option, when the field is redisplayed, blanks are omitted and the field is condensed as shown below:

```
THIS IS A TEST OF THE NEW TED OPTION 'F'.  REMEMBER THAT TED DISPLAYS 80
CHARACTERS ON THE SCREEN.  THREE LEADING BLANKS ARE USED TO INDICATE A
NEW PARAGRAPH.  TEXT FIELD DATA IS ALWAYS STORED EXACTLY AS ENTERED.
WHEN F IS INCLUDED IN THE FORMAT AND THE TEXT FIELD IS REDISPLAYED,
BLANKS ARE OMITTED AND THE FIELD IS CONDENSED. WHEN F IS NOT INCLUDED,
THE FIELD IS REDISPLAYED AS ENTERED.
```

Specifying the Source of Data: The DATA Statement

The DATA statement marks the end of the executable statements in a request. It also specifies the source of the data.

Syntax: How to Use a DATA Statement

```
DATA [ON ddname|VIA program]
```

where:

`ON ddname`

Indicates that the data is in a data source allocated to *ddname*.

`VIA program`

Indicates that the data is supplied directly from another computer program.

Type the DATA statement without parameters if:

- The data comes from the request itself.
- The request contains only PROMPT statements to read data.
- The request does not read any data (this occurs when you use a request to browse through a data source using the NEXT statement).

Reading Selected Portions of Transaction Data Sources: The START and STOP Statements

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

Syntax: How to Use a START Statement

```
START n
```

where:

n

Is the number of the first physical record to be processed by the request.

The syntax for the STOP statement is

```
STOP n
```

where:

n

Is the number of the last physical record to be processed by the request.

The START and STOP statements may appear anywhere in the request.

For example, the following request reads 300 records from a transaction data source (ddname SALDATE) starting from the 201st record until the 500th.

```
MODIFY FILE EMPLOYEE
START 201
STOP 500
```

```
FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA ON FIXSAL
END
```

Note that the numbers are that of physical records, not logical records, and that a request reads four physical records as one logical record. Assume each input record consists of four physical records. For example, if you want the request to read the data source starting from after the first ten transactions, type the START statement as

```
START 41
```

because 10 transactions are made up of 40 physical records.

If you are processing a large transaction data source, you may divide the processing into steps using the START and STOP statements. At the completion of each step, make a backup copy of the data source. If a step is aborted for any reason, you can use the last backup to restore the data source.

These two requests are the same. The first processes transactions 1 to 100,000. The second processes transactions 100,001 to 200,000:

```
MODIFY FILE EMPLOYEE
START 1
STOP 100000
FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA ON FIXSAL
END

MODIFY FILE EMPLOYEE
START 100001
STOP 200000
FIXFORM EMP_ID/9 CURR_SAL/8
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA ON FIXSAL
END
```

Modifying Data: MATCH and NEXT

The MATCH and NEXT statements are the core of MODIFY requests; they are the statements that determine which data source records are added, changed, or deleted. They work by selecting a particular segment instance, then updating or deleting it. They may also add new segment instances.

The MATCH statement selects specific segment instances based on their values. The NEXT statement selects the next segment instance after the current position.

The MATCH Statement

The MATCH statement selects specific segment instances based on their values. It compares one or more field values in the instances with corresponding incoming data values. The action it performs depends on whether there is a segment instance with matching field values.

For example, suppose a MODIFY request was processing this incoming data record in comma-delimited format

```
EMP_ID = 123456789, CURR_SAL = 20000, $
```

and that the request contained this MATCH statement:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH INCLUDE
```

This MATCH statement compares the EMP_ID value of an incoming data record to the EMP_ID values in segment instances:

- If a segment instance has EMP_ID value 123456789, the MATCH statement replaces the CURR_SAL value in the instance with the incoming CURR_SAL value of 20000.
- If there is no instance with the EMP_ID value of 123456789, the MATCH statement creates a new segment instance with the EMP_ID value of 123456789 and a CURR_SAL value of 20000.

Notice that the MATCH statement used each of the two incoming data fields differently. It used the EMP_ID field (specified after the word MATCH) to locate the segment instance (or to prove that it did not exist); it never altered the EMP_ID value in the segment. If it did locate the instance, it replaced the CURR_SAL value in the instance with the value in the incoming data field.

To identify the correct segment instance, the field values that the MATCH statement is searching for must be unique to the instance within its segment chain. For the most common types of segments, types S1 and SH1, the key field value is unique to each instance within its segment chain. This is the value you will usually be searching for.

Note that the MODIFY command cannot update key fields. To update key fields, use the FSCAN facility as described in *Directly Editing FOCUS Databases With FSCAN* on page 389.

Remember from the introduction that FOCUS executes a MODIFY request for every transaction.

Syntax: How to Use a MATCH Statement

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

where:

field1 ...

Are the names of incoming data fields to be compared with similarly named data source fields. The names may be full field names, aliases, or truncations. If a field value is missing, the value is treated as zeros for numeric fields and blanks for alphanumeric fields.

These fields are segment key fields unless the MATCH statement is modifying a segment of type S0 or blank. If the segment is type Sn or SHn and you do not specify the segment keys, the request adds the keys to the list automatically and displays a warning message.

If the list of fields is too long to fit on one line, begin each line with the word MATCH. For example:

```
MATCH EMP_ID DAT_INC TYPE
MATCH PAY_DATE DED_CODE
```

To compare the values of all fields in the data source with incoming values, enter:

```
MATCH *
```

To compare the values of all key fields in the data source with incoming values, enter:

```
MATCH * KEYS
```

To compare the values of all key fields in a particular segment, type

```
MATCH * KEYS SEG n
```

where *n* is either the segment name or number as determined by the ? FDT query (described in the *Developing Applications* manual).

action-1

If the MATCH statement locates a segment instance with a data value matching the incoming data value (ON MATCH), it performs this action.

action-2

If the MATCH statement cannot locate a segment instance with a value matching the incoming data value (ON NOMATCH), it performs this action.

action-3

Whether or not the MATCH statement locates a segment instance with a value matching the incoming data value (ON MATCH/NOMATCH), it performs this action.

Note that you may include many ON MATCH and ON NOMATCH phrases in one MATCH statement. MATCH phrases can precede or follow NOMATCH phrases. The actions you may use in MATCH statements are listed in the section below. They fall into seven groups:

- Actions that modify segments.
- Actions that control MATCH processing.
- Actions that read incoming data fields.
- Actions that perform computations and validations or type messages to the terminal.
- Actions that control Case Logic.
- Actions that control multiple-record processing.
- Actions that activate and deactivate fields.

Please note the following rules regarding the MATCH statement:

- Each phrase of the MATCH statement must start on a separate line.
- The ON MATCH and ON NOMATCH phrases may be reversed.
- If an action has a list of fields, but the list of fields is too long to fit on one line, you may break the list into two or more lines. Begin each line with the ON MATCH or ON NOMATCH phrase, followed by the action. For example:

```
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_SAL
  ON MATCH UPDATE CURR_JOBCODE ED_HRS
```

Syntax: How to Specify Actions With the ON MATCH/NOMATCH Phrase

The MATCH statement has an ON MATCH/NOMATCH phrase. This phrase specifies an action to be taken regardless of whether the field value for which the MATCH statement is searching exists in the data source. This phrase is especially useful when you are using CRTFORMs with display or turnaround fields (see *Designing Screens With FIDEL* on page 227). For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
 "ENTER EMPLOYEE'S ID: <EMP_ID"
MATCH EMP_ID
    ON MATCH/NOMATCH CRTFORM LINE 3
 "ENTER DEPARTMENT: <T.DEPARTMENT"
 "ENTER NEW SALARY: <T.CURR_SAL"
    ON MATCH UPDATE DEPARTMENT CURR_SAL
    ON NOMATCH INCLUDE
DATA VIA FI3270
END
```

This request prompts you for an employee's ID. It then searches for the ID in the data source. It prompts you for the employee's new department and salary, whether the ID is in the data source or not. If the ID is in the data source, it updates the employee's department and salary; otherwise, it adds a new segment instance with the information.

You could not have placed the CRTFORM statement before the MATCH statement, because the CRTFORM statement contains turnaround fields.

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

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

Note: TED in MODIFY can be used only with fields that have a text (TX) format (see *Entering Text Data Using TED* on page 69 for entering and editing text fields with TED).

Reference: MATCH Statement Defaults

The following are defaults affecting the MATCH statement:

- If a MODIFY request has neither MATCH nor NEXT statements, it defaults to:

```
MATCH *
ON NOMATCH INCLUDE
```

It adds the instance even if another instance has the same key values. Since key values uniquely identify segments, you should avoid doing this unless you are loading data into a newly created data source, the incoming data is in a data source, and you know that there are no duplicate key values in the data.

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

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9   LAST_NAME/15  FIRST_NAME/10
FIXFORM PAY_DATE/I6     GROSS/D12.2
DATA ON EMPDATA
END
```

- If a MATCH statement has neither an ON MATCH nor an ON NOMATCH phrase, the MATCH statement defaults to:

  ```
  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 phrase:

  ```
  ON NOMATCH REJECT
  ```

Adding, Updating, and Deleting Segment Instances

The most important function of the MATCH statement is the adding, updating, and deleting of segment instances. The MATCH statement does this by first searching a particular segment chain within a segment for specific instances (segment chains are groups of segment instances associated with an instance in the parent segment). The root segment contains just one segment chain; descendant segments are composed of many segment chains. How the MATCH statement selects segment chains in descendant segments is explained in *Modifying Data: MATCH and NEXT* on page 75.

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:

`ON NOMATCH INCLUDE`	The instance is not yet in the segment. Therefore, the request creates a new instance using values in the transaction.
`ON MATCH REJECT`	The new instance already exists in the segment. Therefore, the request does not add the instance to the data source. Rather, it rejects the transaction.
`ON MATCH UPDATE` *`list`*	The instance exists in the segment. Therefore, the request changes the values of the data source 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.

Example: Adding Segment Instances

The syntax of a MATCH statement that adds segment instances is:

```
MATCH keyfield
  ON MATCH REJECT
  ON NOMATCH INCLUDE
```

When you include a new instance, the request fills the instance with the transaction field values. If some segment fields are absent in the transaction, they become blank or zeros in the instance, or the MISSING symbol if the field is described with the MISSING=ON attribute (discussed in the *Describing Data* manual).

FOCUS determines the placing of the instance within a segment chain based on the current position. The current position is the position of the instance you last added to the chain.

When FOCUS adds the next instance to a keyed segment, it determines whether the instance goes before or after the current position based on the sort order of the segment. If the instance goes after the current position, FOCUS matches field values from the current position forward until it finds the proper place for the new instance. If the instance goes before the current position, FOCUS matches field values from the beginning of the chain forward until it finds the place for the new instance.

To increase efficiency, submit your transactions in the same sorted order as the segment (ascending order for S*n* segments, descending order for SH*n* segments). This causes FOCUS to move through the chain in one direction only.

If you do not submit the transactions in sorted order, you may get this message:

```
WARNING..TRANSACTIONS ARE NOT IN SAME SORT ORDER AS FOCUS FILE
PROCESSING EFFICIENCY MAY BE DEGRADED
```

This condition indicates that data will not be loaded in an optimal manner.

The following request adds new instances to the root segment of the EMPLOYEE data source. The fields EMP_ID (the key field), LAST_NAME, and FIRST_NAME in the new instances are filled with incoming data values; the other fields are left zero or blank:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee's ID, last name, and first name.
2. You enter ID 071382660, last name SMITH, and first name HENRY.
3. The request determines whether ID 071382660 is in the segment. It is there, so the request rejects the transaction, displaying a message telling you so.
4. The request prompts you again for an employee's ID, last name, and first name.
5. You enter ID 123456789, last name SMITH, and first name HENRY.

6. The request determines whether ID 123456789 is in the segment. It is not there, so the request adds a new segment instance, with 123456789 as the key value, SMITH in the LAST_NAME field, and HENRY in the FIRST_NAME field. All other fields in the instance are blanks and zeros.

Example: Updating Segment Instances

The syntax of a MATCH statement to update segment instances is

```
MATCH keyfield
  ON MATCH UPDATE list
  ON NOMATCH REJECT
```

where *list* is a list of data source fields to be updated using the values in the transaction. If the list of fields is too large to fit on one line, begin each line with the ON MATCH UPDATE phrase. For example:

```
ON MATCH UPDATE EMP_ID LN FN
ON MATCH UPDATE HDT DPT CSAL
ON MATCH UPDATE CJC OJT
```

To update all fields in a matched segment (except the key fields), type:

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

Note: You cannot update key fields. To change key fields, use the FSCAN facility as described in *Directly Editing FOCUS Databases With FSCAN* on page 389.

The following request updates the salary (CURR_SAL field) for employees you specify:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee's ID and a new salary.
2. You enter ID 123123123 and a salary of $20,000.
3. The request searches the segment for ID 123123123 but cannot find the value. It rejects the transaction.
4. The request prompts you again for an employee ID and new salary.
5. You enter ID 071382660 and a salary of $20,000.

6. The request finds ID 071382660 in the segment and changes the employee's salary to $20,000.

You can combine adding and updating operations in one MATCH statement:

```
MATCH keyfield
  ON MATCH UPDATE field-1 field-2 ... field-n
  ON NOMATCH INCLUDE
```

This statement searches for a segment instance with a key field value the same as the similarly named incoming field value. If it finds the instance, it updates the instance. If it cannot find the instance, it adds a new instance. For example:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH INCLUDE
```

Example: Deleting Segment Instances

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

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

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

This request deletes records of employees who have left the company:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON MATCH DELETE
  ON NOMATCH REJECT
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee ID.
2. You enter ID 987654321.
3. The request cannot find ID 987654321 in the segment, so it rejects the transaction, displaying a message telling you so.
4. The request prompts you for another employee ID.
5. You enter ID 119329144.

6. The request finds ID 1193291and so on44 and deletes all record of the employee from the data source. This includes the employee's instance in the root segment and all descendant instances (such as pay dates, addresses, and so on).

Performing Other Tasks Using MATCH

You may specify actions in MATCH statements that can stand alone as statements elsewhere in the MODIFY request. These actions are: read incoming data, perform computations and validations, type messages, control Case Logic and multiple record processing, and activate and deactivate fields.

Note that the MATCH statement can perform several actions if the ON MATCH or ON NOMATCH condition occurs. To specify this, assign each action a separate ON MATCH or ON NOMATCH phrase. For example:

```
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE "EMPLOYEE ID NOT FOUND"
  ON NOMATCH REJECT
```

There are two ON NOMATCH phrases in this request: one specifies the TYPE action, the other the REJECT action. If you include a REJECT action, it must appear last; otherwise the request will terminate and generate an error message.

Reference: Reading Data

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

`FIXFORM` *`list`*	Where *list* is a list of fields and formats. Reads in data from a fixed-format data source.
`FREEFORM` *`list`*	Where *list* is a list of incoming data fields. Reads in data from a comma-delimited data source.
`PROMPT` *`list`*	Prompts the user for data in fields named in *list* one field at a time.
`CRTFORM`	Prompts the user for data using the full-screen FIDEL facility. FIDEL is described in *Designing Screens With FIDEL* on page 227.
`TED`	Opens a temporary file for text field data entry using TED.

Reference: Computations, Validations, and Messages

The following actions perform calculations and validations and type messages. These actions work the same as the COMPUTE, VALIDATE, and TYPE statements:

`COMPUTE`	Performs computations.
`VALIDATE`	Performs validations.
`TYPE [ON` *`ddname`*`]`	Types messages to the terminal. When the ON ddname option is used, the messages are sent to a file defined by *ddname*.

Reference: Controlling Case Logic

The following actions control Case Logic. They are discussed in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 149:

`GOTO` *`casename`*	Branches to another case named by *casename*.
`PERFORM` *`casename`*	Branches to another case named by *casename*, then returns to the PERFORM.
`IF` *`expression`* `[THEN]` `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*.

Reference: Controlling Multiple Record Processing

These actions control multiple-record processing and are described in *The REPEAT Method* on page 170:

`REPEAT`	Begins a REPEAT statement that executes a group of MODIFY statements repeatedly.
`HOLD` *`list`*	Where *list* is a list of data fields. Stores field values in a buffer.

Reference: Activating and Deactivating Fields

These actions activate and deactivate fields as described in *Active and Inactive Fields* on page 204:

`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 run them only when the request can locate incoming values in the data source (or confirm that incoming values are not in the data source). This improves efficiency and makes the request logic more flexible.

Example: Using MATCH Actions in a Request

For example, assume you are designing a request to update employee salaries. Those employees who have spent more than 100 hours in class (the ED_HRS field) are granted an extra 3% bonus.

The particular data source you are updating only contains the records of a small number of company employees, but the transaction data source contains records for every employee in the company. If you place the COMPUTE statement calculating the bonuses by itself, it will calculate the bonus for every record in the transaction data source, whether or not the record will be accepted into the data source. Instead, use the COMPUTE statement as an ON MATCH option in a MATCH statement. COMPUTE will then calculate the bonus only for employees in the data source. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
     CURR_SAL = IF D.ED_HRS GT 100 THEN CURR_SAL*1.03
                ELSE CURR_SAL;
  ON MATCH UPDATE CURR_SAL
DATA
```

Note the use of a D. prefixed field in the COMPUTE expression (D.ED_HRS). This field refers only to ED_HRS values in the data source. You may refer to data source fields when using statements in MATCH and NEXT statements or after them. The data source fields must either be in the segment instance you are modifying or in a parent instance along the segment path.

Modifying Segments in FOCUS Structures

This section discusses how the MATCH command modifies segments other than the root segment. The section covers:

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

Reference: Modifying Unique Segments

Unique segments are segments that consist of only one instance for every parent instance. They are always descended from other segments, but may not have descendants themselves. Because unique segment instances are extensions of their parent instances, they have no key fields.

There are two methods of modifying unique segments:

- The CONTINUE TO method allows you to add, update, and delete unique segment instances.
- The WITH-UNIQUES method allows you to add and update unique segment instances, but not to delete them. However, the WITH-UNIQUES method is easier to use.

Syntax: How to Modify Segment Instances Using the CONTINUE TO Method

The CONTINUE TO method first locates the parent instance, then proceeds to the unique instance. The syntax of the MATCH command to modify unique segment instances using the CONTINUE TO method is:

```
MATCH keyfield
  ON NOMATCH action-1
  ON MATCH CONTINUE TO u-field
  ON MATCH action-2
  ON NOMATCH action-3
```

where:

keyfield

Is the key field of the parent segment instance.

action-1

Is the action the request performs if the parent instance cannot be found.

u-field

Is the name of any field in the unique child segment.

action-2

Is the action the request performs if a unique child instance exists.

action-3

Is the action the request performs if a unique child instance does not exist.

The actions that the request can perform are the same as those described in *Adding, Updating, and Deleting Segment Instances* on page 79 and *Performing Other Tasks Using MATCH* on page 84. The MATCH and NOMATCH phrases that follow the ON MATCH CONTINUE TO phrase can be in either order.

This example illustrates how the request selects unique segment instances. The root segment of the EMPLOYEE data source, called EMPINFO, which contains employee IDs, has a unique child segment called FUNDTRAN that contains information on employee bank accounts where pay checks are to be directly deposited. Every EMPINFO instance that describes an employee with a direct deposit bank account has one child instance in the FUNDTRAN segment.

You could prepare the following MODIFY request to enter information on employees that just opened a direct-deposit account:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts for an employee ID, bank name, and bank account number.
2. You enter employee ID 456456456, bank name BEST BANK, and bank account no. 235532.

3. The request does not find employee ID 456456456, so it rejects the transaction.
4. The request prompts you for another employee ID, bank name, and bank account number.
5. You enter employee ID 071382660, bank name BEST BANK, and bank account no. 235532.
6. The request finds ID 071382660. This employee has a segment recorded in the FUNDTRAN segment, meaning that the employee already has a direct-deposit bank account. The request rejects the transaction.
7. The request prompts you for another employee ID, bank name, and bank account number.
8. You enter employee ID 112847612, bank name BEST BANK, and bank account 235532.
9. The request finds employee ID 112847612 but finds no instance recorded for the employee in the FUNDTRAN segment.
10. The request records the bank name and bank account number in a new instance in the unique segment.

The following request updates direct-deposit account information:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH UPDATE BANK_NAME BANK_ACCT
  ON NOMATCH REJECT
DATA
```

The following request deletes account information for employees who have closed their direct-deposit accounts:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO BANK_NAME
  ON MATCH DELETE
  ON NOMATCH REJECT
DATA
```

To modify multiple unique children of one instance using the CONTINUE TO method, use Case Logic as explained in *Case Logic Applications* on page 159.

Syntax: How to Process Unique Instances Using the WITH-UNIQUES Method

The WITH-UNIQUES method processes unique instances as extensions of their parents; that is, it considers a parent instance and its unique child as one instance. This method first searches for the parent instance. If it finds the parent, it can update the parent instance and create or update the unique child at the same time. If it does not find the parent, it can create the parent instance and the unique child at the same time.

The syntax for the MATCH statement using the WITH-UNIQUES method is

```
MATCH WITH-UNIQUES keyfield
  ON MATCH action1
  ON NOMATCH action2
```

where:

keyfield

Is the key field in the parent segment.

action1

Is the action performed if the MATCH statement locates the parent instance.

action2

Is the action performed if the MATCH statement does not locate the parent instance.

The MATCH statement can specify these actions:

- The INCLUDE action, which creates a new parent instance and unique children instances for which there is incoming data.
- The UPDATE action, which updates a parent instance and its unique children. If a child instance does not exist, FOCUS creates one.
- The DELETE action, which deletes the parent instance and all children instances.
- Actions that perform the functions listed in *Performing Other Tasks Using MATCH* on page 84.

Note that the WITH-UNIQUES method can add and update unique instances, but it cannot delete them without deleting the parent instance. To delete unique instances, use the CONTINUE TO method described in *How to Modify Segment Instances Using the CONTINUE TO Method* on page 87.

This MODIFY request adds information on new employees, including information on direct-deposit bank accounts. If an employee is already recorded in the data source, the request rejects the entire transaction. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID FIRST_NAME LAST_NAME
PROMPT BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

This MODIFY request updates employees' account information. If an employee just opened a direct-deposit account, the request automatically creates a new unique instance to record the information. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE BANK_NAME BANK_ACCT
DATA
```

This request adds and updates employees' account information, whether or not the employees are new:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID LAST_NAME FIRST_NAME
PROMPT BANK_NAME BANK_ACCT
MATCH WITH-UNIQUES EMP_ID
  ON NOMATCH INCLUDE
  ON MATCH UPDATE BANK_NAME BANK_ACCT
DATA
```

Note that the WITH-UNIQUES method allows you to include and update the multiple unique children of one instance in one MATCH statement.

When using MATCH WITH-UNIQUES followed by ON MATCH COMPUTE, each computed field must have its own ON MATCH COMPUTE statement.

Modifying Segments

The following examples show how to modify segements.

Example: Modifying Descendant Segments

Modifying descendant segments is similar to modifying the root segment, with one difference: when a MATCH statement searches a root segment for a key field value, it searches every instance of the segment. When the MATCH statement searches a descendant segment, however, it searches only the segment chain belonging to a particular parent instance. If the MATCH statement cannot find the key field value in this chain, it executes the ON NOMATCH phrase. To modify the chain, you must first identify the parent instance using a previous MATCH statement.

The following example illustrates this. The EMPLOYEE data source contains two segments: An EMPINFO segment containing employee IDs, and a child segment called SALINFO that keeps track of each employee's monthly pay. Each of these IDs has an instance in the SALINFO segment for each month that the employee worked (for example, an employee working for eight months has eight instances in the SALINFO segment).

To modify a June instance in the SALINFO segment, you must first identify which employee was paid in June. If the MODIFY request cannot find the June instance for one employee, it will execute the ON NOMATCH phrase even though a June instance exists for another employee.

This request adds a new monthly pay instance for each employee in the company. Note the word CONTINUE, which causes the request to proceed to the next MATCH statement (which adds the instances to the descendant segment) without taking any action. Also note that the phrase ON NOMATCH CONTINUE is illegal:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

An execution might go as follows:

1. The request prompts you for an employee ID, the date the employee was paid, and the gross earnings paid.
2. You enter an employee ID 159159159, pay date 820831 (August 31, 1982), and gross earnings of $916.67.
3. The request cannot find ID 159159159, so it rejects the transaction.
4. The request prompts you for another employee ID, pay date, and gross earnings.
5. You enter employee ID 071382660, pay date 820831, and gross earnings of $916.67.

6. The request finds ID 071382660, and searches the SALINFO segment chain belonging to 071382660 for the pay date 820831.
7. The request finds the pay date 820831 in the segment chain. Since the instance already exists, the request rejects the transaction.
8. You enter employee ID 071382660, pay date 820930 (September 30, 1982), and gross earnings of $916.67.
9. The request finds ID 071382660, and searches the SALINFO segment chain belonging to 071382660 for the pay date 820930.
10. The request does not find pay date 820930 in the segment chain, so it includes a new instance in the SALINFO segment chain for pay date 820930 with gross earnings of $916.67.

If your request prompts for data (using either PROMPT or CRTFORM), it is better to prompt for the child key field values after the request locates the parent key field values. This spares the user from typing the child key if the request cannot locate the parent key. You can rewrite the previous request as:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT PAY_DATE GROSS
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

You can also write the request to include a new EMPINFO segment instance and a new SALINFO instance if the employee's ID is not already there:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
  ON NOMATCH INCLUDE
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON NOMATCH INCLUDE
  ON MATCH REJECT
DATA
```

The first MATCH statement searches the EMPINFO statement for the employee ID that you entered. If it does not find the ID, the request creates a new EMPINFO segment instance with the new ID, and a descendant SALINFO instance with the pay date and monthly pay you entered.

Note that when an INCLUDE action creates a new segment instance, it also creates all descendant instances for which data is present.

If the employee ID is already in the data source, the second MATCH statement searches the SALINFO segment for the pay date you entered. If it does not find the ID, the request creates a new SALINFO instance with the pay date. If the pay date is already in the segment, the request rejects the transaction.

This request updates monthly pay instances:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT PAY_DATE GROSS
MATCH PAY_DATE
  ON MATCH UPDATE GROSS
  ON NOMATCH REJECT
DATA
```

This request deletes monthly pay instances:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT PAY_DATE
MATCH PAY_DATE
  ON MATCH DELETE
  ON NOMATCH REJECT
DATA
```

You may combine the MATCH statements in the request into one statement. This is called matching across segments. To match across segments, specify the key fields that the request must search for from the root segment down to the descendant segment (in that order) after the MATCH keyword. For example, the request above that updates employee's monthly pay can be rewritten this way:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID PAY_DATE
  ON NOMATCH REJECT
  ON MATCH UPDATE GROSS
DATA
```

This is the request shown earlier in this section that adds data on new employees and employees' monthly pay:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID
  ON MATCH CONTINUE
  ON NOMATCH INCLUDE
MATCH PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

This request can be rewritten this way:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE GROSS
MATCH EMP_ID PAY_DATE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Note: When a MATCH statement matches across segments, the explicit ON MATCH and ON NOMATCH phrases in the statement are only executed for the last descendant segment (key field PAY_DATE in the example). For the other segments, the request executes default phrases. If you are updating or deleting instances, these phrases are:

```
ON MATCH CONTINUE
ON NOMATCH REJECT
```

If, for example, you include an ON NOMATCH TYPE phrase in the MATCH statement, the phrase only types a message when there is an ON NOMATCH condition on the last segment.

If you are adding new instances, the default phrases are:

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

Because of these defaults, use this technique only when you are confident that you understand the logic of the request.

Example: Modifying FOCUS Structures of Three or More Levels

What has been said for two-level FOCUS structures is true for three or more levels. To modify a descendant segment instance, you must first identify the parent instances to which the descendant instance belongs, from the root segment down to the immediate parent segment (the descendant segment instance belongs to a parent instance, that instance belongs to grandparent instance, and so on up the FOCUS structure to one of the root instances).

The following request illustrates this. The SALINFO segment has a child segment called DEDUCT that records all the different deductions that are taken from each monthly wage. If four deductions are taken from a monthly pay, that pay has four instances in the DEDUCT segment. The key field in the DEDUCT segment is DED_CODE; it specifies the type of deduction, such as certain taxes. The amount of the deduction is contained in the field DED_AMT.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE DED_CODE DED_AMT
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DED_CODE
  ON NOMATCH REJECT
  ON MATCH UPDATE DED_AMT
DATA
```

Example: Modifying Sibling Segments (Multi-Path Data Sources)

If you are modifying sibling segments (segments that have a common parent), place the MATCH statements modifying the siblings in any order after the MATCH statement identifying the parent instance. Each sibling must have a separate MATCH statement. If you are modifying descendants of one of the siblings, the MATCH statements that modify the children should follow immediately after the MATCH statement that identifies the sibling.

The following request updates the SALINFO and ADDRESS segments, both children of the EMPINFO segment. The ADDRESS segment contains both home and bank addresses of the employees; its key field is TYPE, which indicates whether the address is a home address or a bank address.

The request is as follows:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT PAY_DATE GROSS TYPE ADDRESS_LN1
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON MATCH UPDATE GROSS
MATCH TYPE
  ON NOMATCH REJECT
  ON MATCH UPDATE ADDRESS_LN1
DATA
```

Syntax: How to Modify Segments With No Keys

Segments of types S0 and blank (SEGTYPE= ,) have no key fields. Segments of type blank are always descendant segments; they can never be root segments. Segments of type S0 can be root segments.

To modify these segments, the MATCH statement selects instances by comparing the values of one or more fields in the segment to a similarly named transaction field. The MATCH statement has the form

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

where:

field-1 ...

Are any fields in the segment you are modifying.

* SEG *n*

Matches all fields in the segment, where *n* is either the segment name or number as determined by the ? FDT query (described in the *Developing Applications* manual).

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

Example: Storing Data With Type S0 Segments

When you add a segment instance to a type S0 segment, FOCUS matches field values in the segment chain from the current position forward through the chain, inserting the instance in the chain based on ascending order. FOCUS does not search the chain from the beginning; therefore, if the instance belongs before the current position, FOCUS inserts the instance at the end of the chain (this means that if you are adding instances to a new segment chain, FOCUS stores the instances in the order of submission). It may insert the instance even if another instance has the same field values and you specified ON MATCH REJECT. If, however, you sort the transactions in ascending sequence before submitting them, you will preserve the correct sequence in the chain. You will also prevent adding duplicate segments unless you specify ON MATCH INCLUDE.

Because it is difficult to ensure that segments of type S0 do not have instances with duplicate field values, they are difficult to maintain. You should only use them for data that needs to be loaded in once and does not need to be changed or deleted.

This is a sample FOCUS data source that stores memos, called MEMO. The Master File is:

```
FILE=MEMO ,SUFFIX=FOC ,$
SEGMENT=MEMOSEG  ,SEGTYPE=S1 ,$
 FIELD=MEMO_NAME ,ALIAS=MEMO ,FORMAT=A25      ,$
SEGMENT=TEXTSEG  ,SEGTYPE=S0 ,PARENT=MEMOSEG  ,$
 FIELD=LINE      ,ALIAS=LN   ,FORMAT=A70      ,$
```

The following request enters ten-line memos into the data source:

```
MODIFY FILE MEMO
PROMPT MEMO_NAME 10 (LINE)
MATCH MEMO_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
MATCH LINE
  ON MATCH INCLUDE
  ON NOMATCH INCLUDE
DATA
```

Note: The INCLUDE action in both ON MATCH and ON NOMATCH phrases adds a line of text even if the line is the same as another line in the memo (which would happen if you have more than one blank line in the memo) in all circumstances.

Reference: Type Blank Segments

When you add an instance to a type blank segment, the MODIFY request compares the instance you are adding to every instance in the segment chain, based on the fields you specify in the MATCH statement. Thus, if you specified the ON MATCH REJECT phrase in the MATCH statement, the request does not allow you to add an instance that has the same field values you are matching on as another instance.

You modify type blank segments the same way you modify other segments. Be careful, however, that the fields you are matching on uniquely identify the segment instances, or you may not be able to select the instance you want to modify. (MODIFY requests always select the first instance that fulfills the match conditions.)

Example: Modifying Segments With Multiple Keys

Segments may have multiple keys. These segments are types Sn or SHn where *n* is the number of keys. For example, a segment in ascending order that has two keys is type S2; that is, it has the attribute SEGTYPE=S2 in the Master File. Multiple keys are necessary when the first field alone cannot uniquely identify a segment instance. For example, a segment has three fields as described by the Master File:

```
FILE=ADDRESS ,SUFFIX=FOC ,$
   SEGMENT=ADDRSEG ,SEGTYPE=S2 ,$
    FIELD=LAST_NAME   ,ALIAS=LNAME  ,FORMAT=A15 ,$
    FIELD=FIRST_NAME  ,ALIAS=FNAME  ,FORMAT=A15 ,$
    FIELD=ADDRESS     ,ALIAS=ADDR   ,FORMAT=A80 ,$
```

Since LAST_NAME field is not enough to identify individual segment instances (some people share the same last name), the segment uses the first two fields, LAST_NAME and FIRST_NAME, as keys.

Note that multiple keys must always be the first fields in the segment, and they must be next to each other; that is, a non-key field cannot be between two key fields.

Modifying segments with multiple key fields is the same as modifying segments with one key field. The one difference is that you must specify all the key fields in the MATCH phrase.

To enter data into the ADDRESS data source, you prepare the following MODIFY request:

```
MODIFY FILE ADDRESS
PROMPT LAST_NAME FIRST_NAME ADDRESS
MATCH LAST_NAME FIRST_NAME
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

A sample execution might go as follows:

1. The request prompts you for the last name, first name, and address.
2. You enter last name FOX, first name GEORGE, and address 2365 N. HAMPTON ST. HAMILTON, MN 55473.
3. The request searches the segment for an instance with both last name FOX and first name GEORGE.
4. The request does not find such an instance, so it creates a new instance for George Fox.

Note that you cannot update any of the key fields.

Syntax: How to Use Alternate File Views

To modify descendant segments, you must first specify the parent segments using a series of MATCH statements. You can modify a descendant segment directly by declaring the segment to be the root segment of an alternate file view. To do this, the segment must fulfill three conditions:

- ❑ The segment must be type S1 or SH1.
- ❑ The key field must be indexed.

- The key field values should be unique throughout the data source.

To declare an alternate file view, you begin the MODIFY request this way

```
MODIFY FILE filename.field
```

where:

filename

Is the name of the FOCUS data source you are modifying.

field

Is the name of the indexed key field in the root segment of the alternate file view.

Note that you can only update the root segment of the alternate file view; you cannot add or delete segment instances. However, you can add, update, and delete segment instances in the descendants of this segment. In addition, you may make use of external indices only using the FIND and LOOKUP functions. Be aware that an external index cannot be used as an entry point. For example,

```
MODIFY FILE filename.field
```

will be ineffective. FIND and LOOKUP are described in *Special Functions* on page 122.

This sample FOCUS data source, called BANK, contains information on bank accounts. The Master File is:

```
FILE=BANK ,SUFFIX=FOC ,$
SEGMENT=CUSTSEG ,$
  FIELD=SOC SEC NUM   ,ALIAS=SSN      ,FORMAT=A9   ,$
  FIELD=NAME          ,ALIAS=NAME     ,FORMAT=A30  ,$
SEGMENT=ACCTSEG ,SEGTYPE=S1 ,PARENT=CUSTSEG        ,$
  FIELD=ACCT NUM      ,ALIAS=ACCOUNT  ,FORMAT=A10  ,
  FIELDTYPE=I                                      ,$
FIELD=AMOUNT          ,ALIAS=AMOUNT   ,FORMAT=D10.2,$
SEGMENT=TRANSSEG ,SEGTYPE=S1 ,PARENT=ACCTSEG       ,$
  FIELD=TRANSNUM      ,ALIAS=TNUM     ,FORMAT=I5   ,$
  FIELD=TRANTYPE      ,ALIAS=TTYPE    ,FORMAT=A1   ,$
  FIELD=TR_AMOUNT     ,ALIAS=TAMOUNT  ,FORMAT=D8.2 ,$
```

This Description contains three segments:

- The CUSTSEG segment contains social security numbers and names of bank depositors.
- The ACCTSEG segment, child of CUSTSEG, contains account numbers and the amount of money in each account. Note that the field ACCT_NUM is indexed and that each account number is unique throughout the data source.

- The TRANSSEG segment, child of ACCTSEG, contains information on individual bank account transactions: the transaction serial number (TRANSNUM), the type of transaction (TRANTYPE, which contains a D for deposits and a W for withdrawals), and the amount of the transaction (TR_AMOUNT).

To add new account information in the BANK data source, prepare the following MODIFY request:

```
MODIFY FILE BANK
PROMPT SSN NAME ACCT_NUM AMOUNT
MATCH SSN
  ON NOMATCH INCLUDE
  ON MATCH CONTINUE
MATCH ACCT_NUM
  ON NOMATCH INCLUDE
  ON MATCH REJECT
DATA
```

The MODIFY request above first specifies the parent segment CUSTSEG (MATCH SSN) before the child segment ACCTSEG (MATCH ACCT_NUM). Since ACCTSEG is an S1 segment with an indexed key field (ACCT_NUM), you can modify the ACCTSEG directly with this request:

```
MODIFY FILE BANK.ACCT_NUM
PROMPT ACCT_NUM AMOUNT
MATCH ACCT_NUM
  ON NOMATCH REJECT
  ON MATCH UPDATE AMOUNT
DATA
```

You may modify the root segment of the alternate file view and its descendants in the original data source structure, but not its parents. In the BANK data source, you may modify the TRANSSEG segment using the above alternate file view but not the CUSTSEG segment.

This request adds information on new bank account transactions to the data source:

```
MODIFY FILE BANK.ACCT_NUM
PROMPT ACCT_NUM AMOUNT PROMPT TRANSNUM TRANTYPE TR_AMOUNT
MATCH ACCT_NUM
  ON NOMATCH REJECT
  ON MATCH UPDATE AMOUNT
MATCH TRANSNUM
  ON MATCH REJECT
  ON NOMATCH INCLUDE
DATA
```

Selecting the Instance After the Current Position: The NEXT Statement

The NEXT statement selects the next segment instance after the current position, making the instance the new current position. The current position depends on the execution of MATCH and NEXT statements:

- If a MATCH or NEXT statement selects a segment instance, the instance becomes the current position within the segment.
- If a MATCH or NEXT statement selects a parent instance of a segment chain, the current position is before the first instance in the chain.
- At the beginning of a request, the current position in the root segment is before the first instance.

The NEXT statement can modify segment instances similarly to the MATCH statement and follows the same rules (see *The MATCH Statement* on page 75). However, the NEXT statement is most often used for displaying data source values.

Syntax: How to Use a NEXT Statement

The syntax of the NEXT statement is

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

where:

field

Is any field in the segment whose instances are being selected.

action-1

Is the action the request takes if there is a next instance to select.

action-2

Is the action the request takes if it has reached the end of the segment chain.

There can be many ON NEXT and ON NONEXT phrases in a single NEXT statement. Each phrase specifies one action.

An action can be any action that is legal in the MATCH statement (see *Adding, Updating, and Deleting Segment Instances* on page 79 and *Performing Other Tasks Using MATCH* on page 84). However, use ON NEXT INCLUDE and ON NONEXT INCLUDE phrases only to add instances to segments of type S0 or blank. If you use these phrases to modify other segments, you may duplicate what is already there. The difference between the two phrases is:

- ON NEXT INCLUDE adds a new segment instance after the current position.
- ON NONEXT INCLUDE adds a new instance at the end of the segment chain. The phrase ON NEXT INCLUDE is only valid for segments with type S0 or blank.

The following phrases are always illegal:

```
ON NONEXT UPDATE
  ON NONEXT DELETE
  ON NONEXT CONTINUE
  ON NONEXT CONTINUE TO
```

This phrase is legal even in requests that do not involve Case Logic:

```
ON NONEXT GOTO EXIT
```

The phrase terminates the request when the NEXT statement reaches the end of the segment chain.

Note that a NEXT statement can have multiple ON NEXT and ON NONEXT phrases. For example, the following statement displays the salaries of every employee in the data source and shows what their salaries would be if they are granted a 5% increase:

```
NEXT EMP_ID
  ON NEXT COMPUTE NEWSAL = 1.05 * D.CURR_SAL;
  ON NEXT TYPE
     "EMPLOYEE <D.EMP_ID SALARY NOW:<D.CURR_SAL"
     "SALARY PLUS 5% INCREASE: <NEWSAL"
  ON NONEXT TYPE
     "END OF EMPLOYEE FILE"
  ON NONEXT GOTO EXIT
```

Example: Selecting Instances

You can use NEXT statements in non-Case Logic requests to modify or display the data in:

- The entire root segment.
- The first instances of segment chains in descendant segments.

To modify or display data in *entire* descendant segment chains, you must use Case Logic as described in *Case Logic Applications* on page 159.

The NEXT statement can modify and display data in the root segment. This request displays all the employee IDs in the employee ID segment:

```
MODIFY FILE EMPLOYEE
NEXT EMP_ID
  ON NEXT TYPE "EMPLOYEE ID: <D.EMP_ID"
  ON NONEXT GOTO EXIT
DATA
```

When a NEXT statement modifies or displays data in a descendant segment, it can do so only to the first instance in a segment chain. Consider the following request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE "YOU ENTERED ID <EMP_ID"

NEXT PAY_DATE
  ON NEXT TYPE
     "THIS EMPLOYEE'S LAST PAY DATE"
     "WAS <D.PAY_DATE"
  ON NONEXT GOTO EXIT
DATA
```

The MATCH statement selects an instance with a particular employee ID. The NEXT statement selects the instance with the employee's last pay date (the pay dates are organized in the data source from high to low). The PAY_DATE segment is a child of the EMP_ID segment.

The NEXT statement is at its most powerful when it is used to browse through an entire chain. To browse through a chain in a descendant segment, you must use Case Logic, as described in *Case Logic Applications* on page 159.

Displaying Unique Segments

You can use the NEXT statement to display and modify the contents of unique segments using two methods (see *Modifying Segments in FOCUS Structures* on page 87):

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

Syntax: How to Use the CONTINUE TO Method

The syntax of the CONTINUE TO method is

```
NEXT field
  ON NONEXT action-1
  ON NEXT CONTINUE TO u-field
     ON NEXT action-2
     ON NONEXT action-3
```

where:

field

Is the first field in the parent instance.

action-1

Is the action the request performs if there are no more instances in the parent segment chain.

u-field

Is the name of any field in the unique child segment.

action-2

Is the action the request performs if the parent instance has a unique child instance.

action-3

Is the action the request performs if the parent instance does not have a unique child instance.

Syntax: How to Use the WITH-UNIQUES Method

The syntax of the WITH-UNIQUES method is

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

where:

field

Is the name of any field in the parent segment.

action1

Is the action the request performs if there are no more instances in the chain.

action2

Is the action the request performs if there is a next instance in the chain. This action can be performed on either the parent instance or the unique instance. If an UPDATE action updates a unique instance that does not exist yet, FOCUS creates the instance.

Computations: COMPUTE and VALIDATE

The MODIFY command provides two facilities that perform calculations on incoming data fields, data source fields, and temporary fields. These are:

- The COMPUTE statement. This statement allows you to modify incoming data field values and to define temporary fields.
- The VALIDATE statement. This statement allows you to reject transactions that contain unacceptable values.

FIND and LOOKUP functions can be used only in COMPUTE and VALIDATE statements. For more information, see *Special Functions* on page 122.

Computing Values: The COMPUTE Statement

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

A transaction data source (whether stored on the computer or typed on paper) used to modify a data source often does not contain the same data that is to go into the data source fields. There are many reasons for this:

- The incoming data contains short codes representing the alphanumeric data that is to go into the data source. For example, incoming records contain the code P for PRODUCTION and M for MIS. The PRODUCTION and MIS values update the DEPARTMENT field.
- The incoming data is repetitive: the same value is used to update each instance or the same series of values is used to update each segment chain. For example, all employees are to receive a pay increase of 5%.
- The incoming data values are calculable from other values. For example, an employee's percentage salary increase is equal to the new salary divided by the old salary minus 1.
- Some values vary in predictable ways depending on other values. For example, employee salary increases depend on the employees' department assignment.

The COMPUTE statement gives you control over the data that modifies the data source. Using COMPUTE you can:

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

The COMPUTE statement works by setting either an incoming data field or a temporary field to the value of an expression. The expression may involve existing data source fields, other temporary fields, and constants.

Note that there are three different types of fields:

- Incoming data fields (also called transaction fields) contain data read from transaction data sources or a terminal. These fields are specified by the FIXFORM, FREEFORM, PROMPT, and CRTFORM statements. They remain incoming data fields even if their values are changed by COMPUTE statements.
- Data source fields contain data stored in the data source. Their field names are prefaced by the D. prefix.
- Temporary fields are created by and receive their values from COMPUTE statements.

The following request uses all three types of fields. The request awards a bonus of $150 to employees who received salary raises:

```
    MODIFY FILE EMPLOYEE
1.  PROMPT EMP_ID CURR_SAL
    COMPUTE
2.    BONUSAL/D8.2 = CURR_SAL + 150;
    MATCH EMP_ID
      ON NOMATCH REJECT
      ON MATCH COMPUTE
3.  CURR_SAL = IF CURR_SAL GT D.CURR_SAL
                THEN BONUSAL
                ELSE CURR_SAL;
      ON MATCH UPDATE CURR_SAL
    DATA
```

The numbers above refer to these fields:

1. The EMP_ID and CURR_SAL fields are incoming data fields, because they are read by a PROMPT statement.
2. The BONUSAL field is a temporary field, because it is created by and receives its value from a COMPUTE statement.

3. The D.CURR_SAL field is a data source field, since its field name is prefaced with the D. prefix.

You may use COMPUTE statements to adjust the values of incoming data fields. For example, your MODIFY request reads salary values from a data source and places them into the field SALARY. You want to increase all these values by 10%. To do so, add this statement to the request:

```
COMPUTE SALARY = SALARY * 1.1;
```

In cases where the same field name exists in more than one segment, and that field must be redefined, the REDEFINES command should be used.

You may use the COMPUTE statement to define an unlimited number of temporary fields. For example, you define a temporary field TEMPSAL to contain the number 25000 if an employee is in the MIS department and the number 18000 if an employee is in the PRODUCTION department:

```
COMPUTE
  TEMPSAL =IF DEPARTMENT IS 'MIS' THEN 25000
      ELSE IF DEPARTMENT IS 'PRODUCTION' THEN 18000;
```

Note that MODIFY requests allow the use of up to 3,072 fields within the request. The number includes:

- Data source fields referred to in the request.
- Temporary fields created by COMPUTE and VALIDATE statements.
- Temporary fields created automatically by FOCUS. These include:
  - FOCURRENT for MODIFY requests run in Simultaneous Usage mode. FOCUS creates one FOCURRENT variable per request.
  - REPEATCOUNT for MODIFY requests containing REPEAT statements. FOCUS creates one REPEATCOUNT variable per request regardless of the number of REPEAT statements.
  - HOLDCOUNT and HOLDINDEX for MODIFY requests containing HOLD statements. FOCUS creates one HOLDCOUNT and one HOLDINDEX variable per request regardless of the number of HOLD statements.

Each field referred to or created in a MODIFY request counts as one field toward the 3,072 total, regardless of how often its value is changed by COMPUTE and VALIDATE statements. However, if a data source field is read by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement and also has its value changed by COMPUTE and VALIDATE statements, it counts as two fields.

FOCUS compiles most COMPUTE and DEFINE calculations when the request is parsed. Typically, the new compilation logic executes the compiled calculations in about one-fifth the time required by uncompiled calculations. However, the compiled form requires more memory. For this reason, very large MODIFY procedures may require more virtual storage to run and, should the MODIFY procedures be compiled, they will occupy more disk space.

There are two places in the MODIFY request where you can use COMPUTE statements:

- At the beginning of the request. COMPUTE statements here define temporary field values for every transaction. Note that these statements may not perform calculations on data source field values (D. fields).
- In or following MATCH and NEXT statements. COMPUTE statements here define temporary field values for transactions depending whether or not the MATCH or NEXT statement selected a particular segment instance. These statements may perform calculations using data source field values.

This section covers:

- The syntax of COMPUTE statements.
- Use of COMPUTE statements in MATCH and NEXT statements.
- Modifying transaction fields.
- Defining non-data source transaction fields.

Syntax: How to Use a COMPUTE Statement

The syntax of the COMPUTE statement is as follows (note that you can place several COMPUTE statements after the COMPUTE keyword):

```
COMPUTE
field[/format] = expression;
field[/format] = expression;
.
.
.
```

where:

field

Is the name of the field being set to the value of *expression*. The field can be an incoming data field or it can be a temporary field (whose name must be different from the incoming field names). Fields can only modify data source fields with the same name.

format

Is the format of the field if the field is temporary. Specify the format when defining the temporary field for the first time. Field formats are described in the *Describing Data* manual.

You can specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

You can specify the YRTHRESH and DEFCENT options to handle cross-century dates. Using these options, and working with cross-century dates, is discussed in the *Developing Applications* manual.

expression;

Is any expression valid in a DEFINE or TABLE COMPUTE statement. In addition, you may use the FIND and LOOKUP functions, described in *Special Functions* on page 122.

Note: The expression can be null; that is, the COMPUTE statement can have the form

```
COMPUTE field/format=;
```

where *format* is the format of the field. This form is used to define transaction fields that are not listed in the Master File.

Note that you must terminate the expression with a semi-colon (;). You may type a COMPUTE statement over as many lines as you need, terminating the expression with a semi-colon. The COMPUTE command supports other attributes such as DFC, YRT, and MISSING. See the *Creating Reports* manual for details.

For example:

```
COMPUTE
CURR_SAL = IF CURR_JOBCODE IS A02 THEN 15000
  ELSE IF CURR_JOBCODE IS B02 THEN 17000
  ELSE IF CURR_JOBCODE IS B12 THEN 18000
  ELSE 20000;
```

In the preceding example, the temporary field CURR_SAL will contain 15000, 17000, 18000, or 20000, depending on the value of CURR_JOBCODE. CURR_SAL will then be used later in the MODIFY request.

You can also place an expression on the same line as a COMPUTE keyword, and several expressions on one line (ending each expression with a semicolon). For example:

```
COMPUTE CURR_SAL=CURR_SAL*1.2; ED_HRS = ED_HRS-5;
```

You can specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

Using the COMPUTE Statement

The following examples show how to use the COMPUTE statement.

Example: Placing COMPUTE Phrases in MATCH and NEXT Statements

You may place COMPUTE statements in MATCH and NEXT statements. The request only performs the computation if the MATCH or NEXT condition is met. These COMPUTE phrases may perform calculations on data source field values if these fields are either in the segment instance being modified or in a parent instance along the segment path (the parent instance, the parent's parent, and so on until the root segment). To specify data source field values (as opposed to values in the transaction field with the same name), affix the D. prefix to the front of the field name.

Note that COMPUTE statements that follow a MATCH or NEXT statement may also perform calculations on data source field values if these fields are in the instance selected by the previous statement (or are in the segment path).

When using MATCH WITH-UNIQUES followed by ON MATCH COMPUTE, each computed field must have its own ON MATCH COMPUTE statement.

The following request calculates employees' new salaries giving them a 10% increase over their present salaries. It only performs this calculations for employees whose IDs are stored in the data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
     CURR_SAL = D.CURR_SAL * 1.1;
  ON MATCH UPDATE CURR_SAL
DATA
```

Example: Changing Incoming Data

You can use the COMPUTE statement to change incoming data. For example, assume you are preparing a MODIFY request to input new salaries into the data source. Just recently, the company granted employees in the MIS department an extra 3% pay raise. Rather than manually recalculating the new salaries for MIS employees, you can include a COMPUTE statement to do it for you:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL DEPARTMENT
COMPUTE
CURR_SAL = IF DEPARTMENT IS 'MIS'
    THEN CURR_SAL * 1.03
    ELSE CURR_SAL;
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

The new salary of employees who work in the MIS department will be 1.03 times more what they would have received ordinarily. Everybody else gets a normal raise.

Syntax: How to Define Non-Data Source Transaction Fields

If the names of incoming data fields are not listed in the Master File describing the data source, you must define them to FOCUS before they are read in by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement. Otherwise, FOCUS rejects the fields as unidentifiable and terminates the request.

To define the fields to FOCUS, specify them with the COMPUTE statement using the notation

```
COMPUTE field/format=;
```

where:

field

Is the incoming data field you want to define to FOCUS.

format

Is the format of the field. Field formats are described in the *Describing Data* manual.

Because there is no expression after the equal sign (=), the request reads the statement before it reads the incoming data. All COMPUTE statements having expressions are executed after the request reads the incoming data.

For example, you want to record promotions to the MIS and Production Departments in the data source. However, the transaction data source you are working with lists the departments by code, not by name: a 1 for MIS and a 2 for Production. You prepare the following MODIFY request:

```
MODIFY FILE EMPLOYEE
COMPUTE DEPCODE/I1=;
PROMPT EMP_ID DEPCODE
COMPUTE
  DEPARTMENT = IF DEPCODE IS 1 THEN 'MIS' ELSE 'PRODUCTION';
MATCH EMP_ID
  ON MATCH UPDATE DEPARTMENT
  ON NOMATCH REJECT
DATA
```

The first COMPUTE statement defines the incoming DEPCODE field to FOCUS. The second COMPUTE statement sets the value of the transaction field DEPARTMENT depending on the value of DEPCODE. This DEPARTMENT field then updates the DEPARTMENT field in the data source.

Compiling MODIFY Expressions Using Native Arithmetic

The native compiler for MODIFY processes COMPUTE, IF, and VALIDATE expressions using the arithmetic operations built into the underlying operating system. This native compiler eliminates internal format conversions and speeds up expression processing. It significantly enhances the speed of expressions that use long packed fields and date fields.

Note: Expression compilers for MODIFY are supported only in Mainframe environments. Linux on the Mainframe does not support these compilers.

Syntax: How to Control Compilation of MODIFY Expressions

```
SET MODCOMPUTE={NATV|NEW|OLD}
```

where:

NATV

Activates the native compiler for MODIFY expressions. NATV is the default value.

NEW

Compiles MODIFY expressions using the standard FOCUS compilation routines, which use high-precision floating point format for all arithmetic operations.

OLD

Does not compile MODIFY expressions.

Reference: Usage Notes for SET MODCOMPUTE

The following are usage notes for SET MODCOMPUTE:

- SET MODCOMPUTE can be issued in a user or system profile or on the command line.
- SET MODCOMPUTE is supported with compiled and uncompiled MODIFY procedures. Expression compilation is different from and compatible with MODIFY procedure compilation.
- Existing compiled MODIFY procedures run without recompilation. The MODCOMPUTE setting has no effect on previously compiled MODIFY procedures. In order to make use of this performance enhancement, compiled MODIFYs must be recompiled with SET MODCOMPUTE=NATV in effect.
- Expressions using the following features are not compiled by the native compiler:

  LIKE operator.

  DEFINE functions.

  LAST function.

Validating Transaction Values: The VALIDATE Statement

Most applications require that data be checked for accuracy before it is accepted into the data source. The VALIDATE statement checks values against certain conditions. If the value fails the test, the request rejects the transaction and displays a warning to the user.

For example, assume you are preparing a MODIFY request to update MIS and Production Department salaries in the data source. No one in those departments is ever paid less than $6,000 per year or more than $50,000. You can use the VALIDATE statement to reject those values that fall outside this range, such as a $700 or a $75,000 salary.

VALIDATE statements work the same way as COMPUTE statements: they set the value of a temporary field to the value of an expression. The only difference is that if the field value is set to 0, FOCUS rejects the transaction being processed and displays this message

```
(FOC421) TRANS n REJECTED INVALID rcode
```

where:

n

Is the number of the transaction being tested.

rcode

Is the variable receiving the test value.

The simplest way to use VALIDATE statements is to have them test the values of incoming data fields. If an incoming value is unacceptable, assign the temporary field a value of 0. Otherwise, assign the field a non-zero value. Note that the temporary field retains its value after the VALIDATE statement, and you may use this value in other calculations.

Tests provided by the DBA functions, which control access to data sources, function as involuntary VALIDATE tests and produce similar error messages.

You can place VALIDATE statements in two places in MODIFY requests:

- At the beginning of the request. VALIDATE statements here test every transaction, discarding those containing invalid values. Expressions in these VALIDATE statements cannot use data source field values (D. fields).
- In MATCH and NEXT statements. VALIDATE statements here test the transaction depending whether or not the MATCH or NEXT statement selected a particular segment instance. Expressions in these VALIDATE statements can use data source field values.

If you are validating fields in a repeating group and one field is rejected, all fields in the repeating group are rejected. However, if you are validating the fields in a MATCH or NEXT statement and one field is rejected, the other fields are not rejected.

If the MODIFY request prompts for data (the PROMPT statement), it is a good idea to validate each field after prompting. If you validate several fields at once, users must enter data for all the fields before the values they enter are tested. If one data value is invalid, they must reenter all the data values. If you validate each field, users will be warned as soon as they enter an invalid value, and the request will reprompt them for the correct value.

This section describes:

- VALIDATE statement syntax.
- Using the VALIDATE statement to validate incoming data.
- Use of the ON INVALID phrase.
- Use of VALIDATE statements in MATCH and NEXT statements.
- Testing for the presence of incoming data.
- Use of the DECODE function in VALIDATE statements.

If you validate data entered on a CRTFORM, invalid values cause the CRTFORM screen to be redisplayed along with the data you entered. This allows you to correct the data and re-enter it. You can deactivate this feature using the DEACTIVATE INVALID feature described in *Active and Inactive Fields* on page 204.

Syntax: How to Use a VALIDATE Statement

The syntax of the VALIDATE statement is as follows (note that you may include several VALIDATE statements after the VALIDATE keyword)

```
VALIDATE
  field[/format] = expression;
  field[/format] = expression;
     .
     .
     .
```

where:

field

Is the name of the temporary field. If this field is set to 0, FOCUS rejects the transaction being processed. Do not use an incoming field name or data source field name for this name.

format

Is the format of the field. The format type must be numeric (I, F, D, or P. Formats are described in the *Describing Data* manual). You need to specify the format only if you will use the field elsewhere in the request.

expression;

Is any expression valid in a DEFINE or TABLE COMPUTE statement (see the *Creating Reports* manual). Also, you may use the LOOKUP and FIND function described in *Special Functions* on page 122. If the value of the expression is 0, FOCUS rejects the transaction being processed. Note that you must terminate the expression with a semicolon (;).

You may specify the MISSING option to declare temporary field values missing if values in the expression are missing. The MISSING option is discussed in the *Creating Reports* manual.

Reference: Using VALIDATE to Test Incoming Data

You use VALIDATE statements most often to test incoming data values, assigning the temporary field a value of 0 if a value is not acceptable. The test expression can span several lines, but it must end with a semi-colon (;). Tests you can use in VALIDATE expressions are:

- IF...THEN...ELSE statements.
- Arithmetic expressions.
- Logical expressions.

- User functions and subroutines.
- DECODE functions.
- FIND and LOOKUP functions (see *Special Functions* on page 122).

You can use IF...THEN...ELSE statements in VALIDATE expressions (up to 16 statements per expression), such as:

```
SALTEST = IF SALARY LT 50000 THEN 1 ELSE 0;
```

If the incoming SALARY value is less than $50,000, the SALTEST temporary field is set to 1. If SALARY is $50,000 or greater, SALTEST is set to 0 and the transaction is rejected. Note that you may use all operations in VALIDATE IF°THEN°ELSE statements that you use in COMPUTE and DEFINE statements (see the *Creating Reports* manual). Also note that all alphanumeric literals must be enclosed in single quotation marks.

Example: Using Logical Expressions

If an expression is evaluated as true, the temporary field is set to 1. Otherwise, the field is set to 0. For example:

```
SALTEST = SALARY LT 50000;
```

Note that you can use AND and OR operands in logical expressions, as discussed in the *Creating Reports* manual. For example:

```
SALTEST = (SALARY LT 50000) AND (JOB EQ 'B12');
```

If the incoming salary value is less than $50,000 and the job code is B12, SALTEST is set to 1. Otherwise, SALTEST is set to 0.

Example: Using the DECODE Function

This function allows you to compare an incoming field value against a list of acceptable and unacceptable values. For example:

```
SALTEST = DECODE JOBCODE (A03 0 B07 0 B12 0 ELSE 1);
```

If the incoming job code is A03, B07, or B12, SALTEST is set to 0.

Example: Using the FIND Function

This function searches another FOCUS data source for the presence of the incoming field value. If the value is there, the temporary field is set to a non-zero value; otherwise the field is set to 0. For example:

```
SALTEST = FIND(EMP_ID IN EDUCFILE);
```

If the incoming employee ID value is not present in the EDUCFILE data source, SALTEST is set to 0. The FIND function is discussed in *Special Functions* on page 122.

The following MODIFY request validates the DEPARTMENT and CURR_SAL fields:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT CURR_SAL
VALIDATE
  DEPTEST = IF DEPARTMENT IS 'MIS' THEN 1 ELSE 0;
  SALTEST = CURR_SAL LT 50000;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
```

This request will only accept your transactions if you enter MIS for the DEPARTMENT field and a value less than 50,000 for the CURR_SAL field.

Syntax: How to Take Action on Invalid Data: The ON INVALID Phrase

If a VALIDATE statement invalidates a transaction, you may take action using the ON INVALID phrase. This phrase allows you to:

- Branch to another case using Case Logic. Case Logic is discussed in *Case Logic* on page 145.
- Type a message. Typing messages are discussed in *Messages: TYPE, LOG, and HELPMESSAGE* on page 130.

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

```
ON INVALID GOTO casename
ON INVALID PERFORM casename
ON INVALID TYPE [ON ddname]
```

where:

`GOTO casename`

Branches to another case called *casename*. GOTO also takes other options described in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 149.

`PERFORM casename`

Branches to another case called *casename*. Execution then continues with the next statement after ON INVALID. PERFORM also takes other options discussed in *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 149.

```
TYPE [ON ddname]
```

Displays a message of up to four lines on the terminal. If you use the ON ddname option, the request writes the message to a sequential data source allocated to *ddname*.

This request updates employee salaries. It warns you when you have entered a salary that fails its validation test:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
VALIDATE
  SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
  ON INVALID TYPE
     "YOU ENTERED A SALARY HIGHER THAN $50,000"
     "THIS SALARY IS TOO HIGH"
     "PLEASE REENTER THE EMPLOYEE ID AND SALARY"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
```

VALIDATE Phrases in MATCH and NEXT Statements

You may place VALIDATE statements in MATCH and NEXT statements. The request only performs the validation if the MATCH or NEXT condition is met. These VALIDATE phrases may use data source fields if these fields are either in the segment instance being modified or in a parent instance along the segment path (the parent instance, the parent's parent, and so on until the root segment). To specify data source field values, affix the D. prefix to the front of the field name.

Note that VALIDATE statements that follow a MATCH or NEXT statement may also use data source fields if these fields are in the instance selected by the previous statement (or are in the segment path).

This request makes sure that an employee's new salary is not less than the present salary after it ascertains that the employee's ID is recorded in the data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT CURR_SAL
  ON MATCH VALIDATE
     SALTEST = IF CURR_SAL GE D.CURR_SAL THEN 1
               ELSE 0;
  ON MATCH UPDATE CURR_SAL
DATA
```

Example: Testing for the Presence of Transaction Data

You may test for missing data values in transactions using the MISSING feature in IF and WHERE phrases, described in the *Creating Reports* manual. These features determine whether an incoming field is present in the transaction or not, and are especially useful when the transactions are in a transaction data source.

This request rejects transactions without a job code:

```
MODIFY FILE EMPLOYEE
FREEFORM EMP_ID CURR_JOBCODE CURR_SAL
VALIDATE
  JOBTEST = IF CURR_JOBCODE IS NOT MISSING THEN 1
            ELSE 0;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_JOBCODE CURR_SAL
DATA
EMP_ID=071382660, CURR_JOBCODE=A13, CURR_SAL=18500.00, $
EMP_ID=112847612,                   CURR_SAL=19200.50, $
END
```

Syntax: How to Validate Values From a List: The DECODE Function

The DECODE function allows you to compare incoming data values against a list of acceptable and unacceptable values. This function is described in the *Creating Reports* manual. This section discusses how best to use the DECODE function to validate data.

The syntax of the DECODE function is

```
field = DECODE fieldname (code1 result1...[ELSE default])
```

where:

field

Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming field name or data source field name for this name.

fieldname

Is the incoming data field being tested.

code1 ...

Is the list of possible values.

result1

Is the number that the temporary field is set to if the incoming field has the preceding value. Place a 0 after invalid values; place a non-zero number after valid values.

ELSE

Indicates what the temporary field is set to if the incoming field does not have a value on the list. This list may have up to 32,767 literals.

For example, you want to record promotions to various company departments in the data source. There are five possible departments: Marketing, Accounting, Shipping, Sales, and Data Processing. You prepare this MODIFY request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DEPARTMENT
VALIDATE
  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:

field

Is the name of the temporary field. If the field is set to 0, the transaction is rejected. Do not use an incoming or data source field name for this name.

infield

Is the incoming field being tested.

ddname

Is the ddname of the file containing the list of possible values. The file may contain up to 32,767 bytes.

m

Is the value of *field* if the incoming data value is not in the list.

Below is a sample DECODE file.

```
MARKETING   1
ACCOUNTING  1
SHIPPING    1
SALES  1
MIS 1
```

Special Functions

There are two functions that you can use only in MODIFY COMPUTE and VALIDATE statements. They are:

- The FIND function, which tests for the existence of indexed values in FOCUS, relational, or Adabas data sources.
- The LOOKUP function, which tests for the existence of non-indexed values in cross-referenced FOCUS, relational, or Adabas data sources and makes these values available for other computations.

Note: The LAST function in MODIFY can be used in COMPUTEs and VALIDATEs, in combination with FREEFORM or FIXFORM, to test incoming transaction values against those from a previously read record. For further information on the LAST function see the *Creating Reports* manual.

Syntax:

How to Test for the Existence of Indexed Values in FOCUS Data Sources: The FIND Function

The FIND function verifies if an incoming data value is in a FOCUS data source field, whether the field is in the data source you are modifying or in another data source. The function sets a temporary field to a non-zero value if the incoming value is in the data source field and 0 if it is not. Note that a value greater than zero confirms the presence of the data value, not the number of instances in the data source field. You can then test and branch on this field using Case Logic, described in *Case Logic* on page 145.

Note that the data source field you are searching must be indexed, and that the FIND function does not work on data sources with different DBA passwords.

The syntax of the FIND function is

```
field = FIND(fieldname [AS dbfield] IN file);
```

where:

field

Is the name of the temporary field.

fieldname

Is the full name (not the alias or a truncation) of the incoming field being tested.

AS *dbfield*

Is the full name (not the alias or a truncation) of the data source field containing values to be compared with the incoming data field. This field must be indexed. If the incoming field and the data source field have the same name, you can omit this phrase.

file

Is the name of the data source.

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

The opposite of FIND is NOT FIND. The NOT FIND function sets a temporary field to 1 if the incoming value is not in the data source and 0 if the incoming value is in the data source. Its syntax is

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

where *field*, *infield*, *dbfield*, and *file* were explained previously.

You can use any number of FIND functions in COMPUTE and VALIDATE statements. However more FIND functions increase processing time and require more buffer space in core.

This request tests if each employee ID entered is also in the EDUCFILE data source. It then displays a message informing you whether it found the ID in the data source or not.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
COMPUTE
  EDTEST = FIND(EMP_ID IN EDUCFILE);
  MSG/A40 = IF EDTEST IS 1 THEN
          'STUDENT LISTED IN EDUCATION FILE' ELSE
          'STUDENT NOT LISTED IN EDUCATION FILE';
MATCH EMP_ID
  ON NOMATCH TYPE "<MSG"
  ON MATCH TYPE "<MSG"
DATA
```

Example: Using the FIND Function in VALIDATE Statements

You may use the FIND function in a VALIDATE statement to test if a transaction field value exists in another FOCUS data source. If the field value is not in that data source, the function returns a value of 0, causing the validation to fail and the request to reject the transaction.

This request updates the number of hours spent by employees in class. It rejects employees not listed in the EDUCFILE data source, which records class attendance:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
  EDTEST = FIND(EMP_ID IN EDUCFILE);
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE ED_HRS
DATA
```

This VALIDATE statement will discard any incoming EMP_ID value not found in the EDUCFILE data source.

Reading Cross-Referenced FOCUS Data Sources: The LOOKUP Function

The LOOKUP function retrieves data values from cross-referenced data sources, both data sources cross-referenced statically in the Master File and data sources joined dynamically by the JOIN command. The LOOKUP function is necessary because, unlike TABLE requests, MODIFY requests cannot read cross-referenced data sources freely. With the LOOKUP function, the requests can use the data in computations and in messages but cannot modify cross-referenced data sources; to modify more than one data source in one request, use the COMBINE command discussed in *Modifying Multiple Data Sources in One Request: The COMBINE Command* on page 196.

The LOOKUP function can read cross-referenced segments that are linked directly to a segment in the host data source (the host segment). This means that the cross-referenced segments must have segment types of KU, KM, DKU, or DKM (but not KL or KLU) or contain the cross-referenced field specified by the JOIN command (see the *Describing Data* manual).

The cross-referenced segment contains two fields of interest:

- The field containing the values you want. This is the field the LOOKUP function specifies. For example, this LOOKUP function retrieves values from the DATE_ATTEND field:

  ```
  RTN = LOOKUP(DATE_ATTEND);
  ```

- The cross-referenced field. This field shares values with a field in the host segment called the host field. These two fields link the host segment to the cross-referenced segment. The LOOKUP function uses the cross-referenced field, which is indexed, to locate a specific segment instance.

To use the LOOKUP function, the MODIFY request reads a transaction value for the host field. The LOOKUP function then searches the cross-referenced segment for an instance containing this value in the cross-referenced field:

- If there are no such instances, the function sets a return variable to 0. If you use the field specified by the LOOKUP function in the request, the field assumes a value of blank if alphanumeric and 0 if numeric.
- If there are instances (there can be more than one if the cross-referenced segment type is KM, DKM, or if you specified the ALL keyword in the JOIN command), the function sets the return variable to one and retrieves the value of the specified field from the first instance it finds.

The syntax of the LOOKUP function is

```
rcode = LOOKUP(field);
```

where:

rcode

Is a variable you specify to receive a return code value. This value is 1 if the LOOKUP function can locate a cross-referenced segment instance, 0 if the function cannot.

field

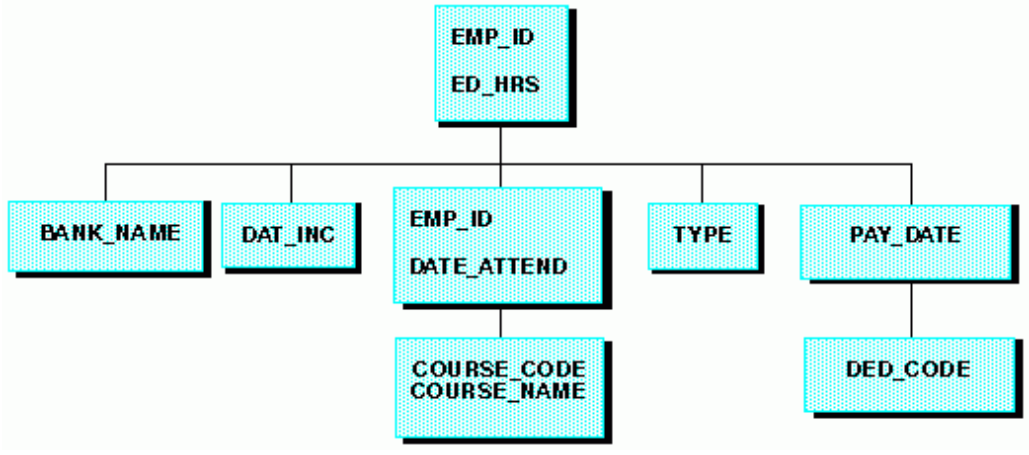
Is the field that you want to retrieve in the cross-referenced data source. Note that this field name cannot exist in the host data source, and that the LOOKUP function may specify only one field at a time. Each field you wish to retrieve requires a separate LOOKUP function. To look up all fields in the cross-referenced segment, use LOOKUP (SEG.field).

Note that there may be no space between LOOKUP and the left parenthesis. The LOOKUP function can exist by itself or as part of a larger expression. If it exists by itself, it must terminate with a semicolon.

For example, you wish to update the amount of classroom hours employees have spent. Because of a new system of accounting, employees taking classes after January 1, 1985 are to be credited with 10% more classroom hours than their records indicate.

The employee IDs (EMP_ID) and classroom hours (ED_HRS) are located in the host segment. The class dates (DATE_ATTEND) are located in the cross-referenced segment. The shared field is the employee ID field.

The data source structure is shown in this diagram:



The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID ED_HRS
COMPUTE
  EDTEST = LOOKUP(DATE_ATTEND);
  COMPUTE
  ED_HRS = IF DATE_ATTEND GE 820101 THEN ED_HRS * 1.1
           ELSE ED_HRS;
MATCH EMP_ID
  ON MATCH UPDATE ED_HRS
  ON NOMATCH REJECT
DATA
```

A sample execution of this request might go as follows:

1. The request prompts you for an employee ID and number of class hours. You enter the ID 117593129 and 10 class hours.
2. The LOOKUP function locates the first instance in the cross-referenced segment containing the employee ID 117593129. Since the instance exists, the function returns a 1 to the EDTEST variable. This instance lists the class date as 821028 (October 28, 1982).
3. The LOOKUP function retrieves the value 821028 for the DATE_ATTEND field.
4. The COMPUTE statement tests the value of the DATE_ATTEND field. Since October 28, 1982 is after January 1, 1982, the statement increases the incoming ED_HRS value from 10 to 11 hours.
5. The request updates the classroom hours for employee 117593129 using the new ED_HRS value.

You may also use a data source value in a specific host segment instance to search the cross-referenced segment. To do this, prepare the request this way:

- In the MATCH statement that selects the host segment instance, activate the host field. This can be done with the ACTIVATE phrase (discussed in *Active and Inactive Fields* on page 204).
- In the same MATCH statement, place the LOOKUP function after the ACTIVATE phrase.

This request displays the employee IDs, dates of salary raises, employee names, and the position each employee held after the raise was granted:

- The employee IDs and names (EMP_ID) are in the root segment.
- The date of raise (DAT_INC) is in the descendant host segment.
- The job titles are in the cross-referenced segment.
- The shared field is JOBCODE. You never enter any job codes; the values are all stored in the data source.

The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID DAT_INC
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
MATCH DAT_INC
  ON NOMATCH REJECT
  ON MATCH ACTIVATE JOBCODE
  ON MATCH COMPUTE
     RTN = LOOKUP(JOB_DESC);
  ON MATCH TYPE
     "EMPLOYEE ID:           <EMP_ID"
     "DATE INCREASE:         <DAT_INC"
     "NAME:  <D.FIRST_NAME   <D.LAST_NAME"
     "POSITION:              <JOB_DESC"
DATA
```

A sample execution might go as follows:

1. The request prompts you for an employee ID and date of pay raise. You enter employee ID 071382660 and date of raise 820101 (January 1, 1982).
2. The request locates the instance containing the ID 071382660, then locates the child instance containing the date of raise 820101.
3. This child instance contains the job code A07. The ACTIVATE statement activates this value, making it available to the LOOKUP function.

4. The LOOKUP function locates the job code A07 in the cross-referenced segment. It returns a 1 into the RTN variable and retrieves the corresponding job description of SECRETARY.
5. The request displays the values using a TYPE statement:

```
EMPLOYEE ID:    071382660
DATE INCREASE:  82/01/01
NAME:           ALFRED STEVENS
POSITION:       SECRETARY
```

Note: You may also need to activate the host field if you are using the LOOKUP function within a NEXT statement. This request, similar to the previous one except for the NEXT statement, displays the latest position held by a particular employee.

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH CONTINUE
NEXT DAT_INC
  ON NONEXT REJECT
  ON NEXT ACTIVATE JOBCODE
  ON NEXT COMPUTE
     RTN = LOOKUP(JOB_DESC);
  ON MATCH TYPE
     "EMPLOYEE ID:          <EMP_ID"
     "DATE OF POSITION:     <DAT_INC"
     "NAME:  <D.FIRST_NAME  <D.LAST_NAME"
     "POSITION:             <JOB_DESC"
DATA
```

Syntax: How to Use an Extended Syntax With LOOKUP

If the function cannot locate a value of the host field in the cross-referenced segment, you may specify that the LOOKUP function locate the next highest or lowest cross-referenced field value in the cross-referenced segment by using an extended syntax.

To use this LOOKUP feature, the index must have been created on FOCUS Release 4.5 or later with the INDEX parameter set to NEW (the binary tree scheme). To determine what type of index your data source uses, enter the ? FDT command (see the *Developing Applications* manual).

Note that a field retrieved by the LOOKUP function does not require the D. prefix to be displayed in TYPE statements. FOCUS treats the field value as a transaction value.

The extended syntax of the LOOKUP function is

```
COMPUTE
  rcode = LOOKUP(field operator);
```

where:

rcode

Is a variable you specify to receive a return code value. (The value the variable receives depends on the outcome of the function below.)

field

Is the name of the field you want to use in MODIFY computations. Note that this cannot be the cross-referenced field.

operator

These parameters specify the action the request takes if there is no cross-referenced segment instance corresponding to the host field value. The actions can be one of the following:

EQ causes the LOOKUP function to take no further action if an exact match is not found. If a match is found, the value of *rcode* is set to 1; otherwise, it is set to 0. This is the default.

GE causes the LOOKUP function to locate the instance with the exact or next highest value of the cross-referenced field.

LE causes the LOOKUP function to locate the instance with the exact or next lowest value of the indexed field.

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

This table summarizes the value of *rcode* depending on which instance the LOOKUP function locates:

Action	rcode value
Exact cross-referenced value located	1
Next highest cross-referenced value located	2
Next lowest cross-referenced value located	-2
Cross-referenced field value not located	0

Reference: Using the LOOKUP Function in VALIDATE Statements

When you use the LOOKUP function, you may want to reject transactions containing values for which there is no corresponding instance in the cross-reference segment. To do this, place the function in a VALIDATE statement. If the function cannot locate the instance in the cross-referenced segment, it sets the value of the return variable to 0. This causes the request to reject the transaction.

The following request updates an employee's classroom hours (ED_HRS). If the employee attended classes on or after January 1, 1982, the request increases the number of classroom hours by 10%. The classroom attendance dates are stored in a cross-referenced segment (field DATE_ATTEND). The shared field is the employee ID.

The request is:

```
MODIFY FIELD EMPLOYEE
PROMPT EMP_ID ED_HRS
VALIDATE
  TEST_DATE = LOOKUP(DATE_ATTEND);
COMPUTE
  ED_HRS = IF DATE_ATTEND GE 820101 THEN ED_HRS * 1.1
           ELSE ED_HRS;
MATCH EMP_ID
  ON MATCH UPDATE ED_HRS
  ON NOMATCH REJECT
DATA
```

If the employee is not recorded in the cross-referenced segment, then the employee has never attended a class. This means that a transaction recording the employee's classroom hours is an error, and should be rejected.

This is the purpose of the LOOKUP function in the VALIDATE statement. If the function cannot locate an employee's record in the cross-referenced segment, it returns a 0 to the TEST_DATE field. This causes the request to reject the transaction.

Messages: TYPE, LOG, and HELPMESSAGE

This section describes how MODIFY requests handle messages. There are four types:

- Messages written into requests.
- Messages indicating the rejection of transactions.
- Messages originating from the Master File with the HELPMESSAGE attribute.
- Messages that echo transactions.

These messages are helpful in debugging MODIFY requests, locating rejected transactions, and instructing the operator. There are two statements and one attribute that control the display of messages:

- The TYPE statement enables you to write messages to the terminal and to sequential files.
- The LOG statement stores incoming or rejected transactions in sequential files and controls the display of rejection messages.
- The HELPMESSAGE attribute is a field attribute included in the Master File (of FOCUS data sources). Text messages specified in the Master are displayed in the TYPE area of MODIFY CRTFORMs.

Displaying Specific Messages: The TYPE Statement

The TYPE statement either appears on the terminal or stores in a sequential file messages that you prepare. This section describes:

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

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

Syntax: How to Use a TYPE Statement

The syntax of the TYPE statement is

```
TYPE [AT START|AT END]  [ON ddname]
"message"
["message"]
```

where:

`AT START`

Displays a message at the beginning of execution only.

`AT END`

Displays a message at the end of execution only. If you are using Case Logic, the TYPE AT END statement must be in the case that generates the end-of-file condition. Either the case must include a FIXFORM or FREEFORM statement that will reach the end of the transaction data source; or a PROMPT statement, at which the user will type END or QUIT; or a CRTFORM statement, at which the user will type END or press the PF3 key.

`ON` *ddname*

Writes the message to a sequential file allocated to *ddname*. The TYPE statement can write lines of up to 256 characters each, including blanks and embedded field values. If you omit this phrase, the request displays the message on the terminal.

message

Is any message. Enclose each line in double quotation marks (except when you want to display two lines as one line, as described later in this section in *Embedding Spot Markers* on page 135.) If you are displaying messages at the terminal, the lines begin in column 2 on the screen. If you are writing the message to a file, the lines begin in column 3 in the file. You may embed spot markers and data fields in the message.

Note that you can type the TYPE statement on one line. For example:

```
TYPE "THIS IS A ONE LINE MESSAGE"
```

TYPE statements can stand by themselves, they can be part of MATCH and NEXT statements, and they can follow VALIDATE statements. For example:

```
MODIFY FILE EMPLOYEE
TYPE
  " "
  "PLEASE ENTER THE FOLLOWING DATA"
  " "
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
DATA
```

This request asks the user to enter data at the beginning of every transaction. Note that there is a blank message line both before and after the message "PLEASE ENTER THE FOLLOWING DATA:" This enhances readability and appearance.

TYPE statements may be part of MATCH and NEXT statements. For example, this request warns the user when an employee ID that the user has entered is not in the data source:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
     " "
     "NO SUCH EMPLOYEE IN THE DATABASE"
     "PLEASE RETYPE THE EMPLOYEE ID"
  ON NOMATCH REJECT
DATA
```

TYPE statements can display messages when incoming data values fail validation tests, as discussed in *Validating Transaction Values: The VALIDATE Statement* on page 114. For example, this request warns the user when a salary higher than $50,000 is entered:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
VALIDATE
  SALTEST = IF CURR_SAL LE 50000 THEN 1 ELSE 0;
  ON INVALID TYPE
     " "
     "THE CURR_SAL VALUE IS OVER 50000"
     "AND THEREFORE CANNOT BE ENTERED INTO THE"
     "DATABASE. PLEASE NOTIFY YOUR SUPERVISOR."
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

Note that ON INVALID TYPE phrases can occur after VALIDATE statements that stand by themselves or are part of MATCH statements. For example:

```
MATCH PAY_DATE
ON NOMATCH REJECT
ON MATCH VALIDATE
  GROSS_TEST = IF GROSS LT 1500 THEN 1 ELSE 0;
  ON INVALID TYPE
     "GROSS OVER $1500. PLEASE REENTER"
```

Reference: Embedding Data Fields

You can embed data fields in the middle of messages. Embedded data fields are described in the *Creating Reports* manual. The kind of field you may embed depends on the position of the TYPE statement:

- TYPE statements preceding MATCH or NEXT statements only accept incoming data fields in messages, not data source fields.
- This request contains a TYPE statement before the MATCH statement:

```
MODIFY FILE EMPLOYEE
FIXFORM EMP_ID/9 X1 CURR_SAL/8
TYPE
   "EMPLOYEE ID: <EMP_ID SALARY: <CURR_SAL"
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA ON EMPSAL
END
```

- TYPE phrases in or following a MATCH or NEXT statement accept both incoming data fields and data source fields in messages. The data source field must either be in the segment instance that the MATCH or NEXT statement is modifying or in a parent instance along the segment path (the parent instance, the parent's parent, and so on to the root segment). To specify a data source field, affix the prefix D. to the field name.

This TYPE phrase displays both the incoming value of CURR_SAL and the data source value:

```
ON MATCH TYPE
   "SALARY ENTERED IS: <CURR_SAL"
   "OLD SALARY WAS: <D.CURR_SAL"
```

You can use embedded fields together in a statement to display a total. This request totals all salaries updated:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
     TOTAL_SAL/D10.2 = TOTAL_SAL + CURR_SAL;
  ON MATCH UPDATE CURR_SAL
TYPE AT END
  "TOTAL OF ALL NEW SALARIES IS <TOTAL_SAL"
DATA
```

Every time the user enters a salary, the request adds it to the running total TOTAL_SAL. After the user enters the last salary, the request displays the TOTAL_SAL value embedded in the message.

Note: Each line of text can contain up to 256 characters. This includes the lengths of the embedded fields as defined by the display field formats (for example, the CURR_SAL field, having the format D12.2M, takes up 15 characters, including decimal point, commas, and dollar sign).

Embedded fields enable you to design your own log files to record transactions, replacing the automatic log file facility activated by the LOG statement. This request logs accepted transactions into the file ACCFILE and logs rejected transactions into the file REJFILE:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH TYPE ON ACCFILE
     "<EMP_ID <12 <CURR_SAL"
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE ON REJFILE
     "<EMP_ID <12 <CURR_SAL"
  ON NOMATCH REJECT
DATA
```

This request records in the ACCFILE file the employee ID and new salary entered by the user if the ID is in the data source and records the ID and salary in the REJFILE file if the ID is not in the data source. Note that the spot markers in both TYPE messages ensure that the fields will be aligned in the files, making the files fixed sequential files. If the request logged the transactions using the MODIFY LOG facility, the files would have been comma-delimited because the request uses PROMPT to input data. Note that you must issue an allocation for each log file prior to using it in the MODIFY request.

Reference: Embedding Spot Markers

You can embed spot markers in TYPE statement messages. Spot markers are devices that place message text at different places on the screen. Spot markers are described in *Tutorial: Painting a Procedure*. Some common spot markers are shown below (where *n* is an integer):

`<n`

Places text starting at the *n*th column.

`<+n`

Places text *n* columns to the right.

`</n`

Places text *n* lines down.

`<0X`

Positions the next character immediately to the right of the last character (skip zero columns). This is used when you have two or more lines between the double quotation marks in a procedure that make up a single line of information on a FIDEL screen. No spaces are inserted between the spot marker and the start of a continuation line.

For example, the statement

```
TYPE
  "THE DOLLAR SIGN IS IN COLUMN 40: <40 $"
  "TEN SPACES ARE EMBEDDED <+10 IN THIS LINE"
  "</1 THIS LINE SKIPS A LINE <0X
  AND PROVIDES AN EXAMPLE OF THE USE <0X
  OF A COLUMN MARKER"
```

produces the following output:

```
THE DOLLAR SIGN IS IN COLUMN 40:       $
TEN SPACES ARE EMBEDDED          IN THIS LINE

THIS LINE SKIPS A LINE AND PROVIDES AN EXAMPLE OF THE USE OF A COLUMN MARKER
```

Note: The spot marker to skip a line, </n, can appear on the same line with other text in a TYPE statement. However, in a CRTFORM, this spot marker must appear on a line by itself (see *Designing Screens With FIDEL* on page 227).

Sometimes, a line of text you want displayed cannot fit on one line within the TYPE command. This can occur because you are indenting lines or because there are non-printable characters in the message, such as spot markers and field prefixes. To have two lines in the TYPE statement displayed as one line, do the following:

- End the first line without an end quotation mark.
- Do not begin the second line with a quotation mark. Instead, begin the line with a <+n spot marker where *n* is any number greater than or equal to zero.

This TYPE statement demonstrates how this feature can be used:

```
TYPE
  "<D.FIRST_NAME <D.LAST_NAME EMP. #<EMP_ID
  <+1 SALARY: <CURR_SAL"
```

If you enter in the employee ID 123764317 and a salary of $27,000, the request displays this message:

```
JOAN IRVING    EMP. #123764317 SALARY: $27,000.00
```

You may write a message of several lines this way. Begin the first line of the message with a quotation mark and end the last line with a quotation mark. Begin alternating lines with the <+1 spot marker. This causes the request to display every two lines of text as one line.

For example, if you type this statement in the request:

```
TYPE
  "SALARY UPDATE PROCEDURE
  <+1 WRITTEN JUNE 26, 1985"
  "ENTER EACH EMPLOYEE ID AND SALARY
  <+1 AFTER THE PROMPTS"
```

The request displays the message as:

```
SALARY UPDATE PROCEDURE WRITTEN JUNE 26, 1985
ENTER EACH EMPLOYEE ID AND SALARY AFTER THE PROMPTS
```

The following request employs both spot markers and embedded fields in messages:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH TYPE
     "</1 EMPLOYEE <EMP_ID NOT IN THE DATABASE"
     "PLEASE RETYPE NUMBER OR NOTIFY SUPERVISOR"
  ON NOMATCH REJECT
  ON MATCH TYPE
     "</1 EMPLOYEE <15 LAST_NAME <30 FIRST_NAME <45 SALARY"
     "</1 <EMP_ID <15 <D.LAST_NAME
     "<+1 <30 <D.FIRST_NAME <40 <D.CURR_SAL"
     "</1 ENTER SALARY FOR EMPLOYEE <EMP_ID"
     " "
  ON MATCH PROMPT CURR_SAL
  ON MATCH UPDATE CURR_SAL
DATA
```

When you run this request, the session looks like this:

```
> EMPLOYEE ON 10/10/98 AT 19.44.47
DATA FOR TRANSACTION    1

EMP_ID    = > 451123478

EMPLOYEE     LAST_NAME   FIRST_NAME   SALARY
451123478    MCKNIGHT    ROGER        $16,100.00

ENTER SALARY FOR EMPLOYEE 451123478

CURR_SAL   = > 18500
DATA FOR TRANSACTION    2

EMP_ID     = >
```

Reference: Screen Attributes

If your request includes CRTFORMs, you can enhance TYPE statements with screen attributes, devices that display a line, part of a line, or an embedded field in color, in reverse video, flashing, or underlined. Screen attributes are discussed in *Designing Screens With FIDEL* on page 227, in connection with the FIDEL facility.

Note the following when using screen attributes in TYPE statements:

- You may use screen attributes only in TYPE statements that follow a CRTFORM and will appear on the screen beneath the CRTFORM during execution.
- Extended attributes in TYPE statements only work on terminals that can process all screen attributes. To use screen attributes in TYPE statements, you must issue the command:

  ```
  SET EXTTERM = ON
  ```

- When you add an attribute to a line, whether you place the attribute before a field or before text, the attribute remains in effect until the end of the line or until the next attribute, whichever comes first.
- Attributes for TYPE statements are cleared at the end of each line. To apply an attribute to a block of text, type the attribute at the beginning of each line.

This request uses attributes in TYPE statements:

```
MODIFY FILE EMPLOYEE
CRTFORM
"ENTER EMPLOYEE ID:     <EMP_ID"
"ENTER SALARY:          <CURR_SAL"

MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
     "<.WHITE. EMPLOYEE #<.AQUA.EMP_ID"
     "<.WHITE. IS <.RED. NOT <.WHITE. IN THE DATABASE"
     "<.WHITE. PLEASE NOTIFY SUPERVISOR"
  ON NOMATCH REJECT
DATA
END
```

The request displays the employee ID in aquamarine and the EMPLOYEE IS NOT IN THE DATABASE message in white, except for the word NOT, which is in red.

Logging Transactions: The LOG Statement

The LOG statement enables you to record transactions in sequential files automatically and to control the display of rejection messages at the terminal. You may use the LOG statement to record transactions in files, one file for each type of transaction: all transactions, accepted transactions, and different types of rejected transactions. The statement can also shut off MODIFY command rejection messages, enabling you to substitute your own.

Syntax: How to Log Transactions in Sequential Files

The LOG statement enables you to record transactions processed by a MODIFY request in sequential files. You can record all transactions or only transactions accepted into the data source. You can record in separate files transactions rejected because of an ON MATCH REJECT or ON NOMATCH REJECT phrase, transactions rejected because of validation tests, and transactions rejected because of format errors.

Note that you can design your own log files by using the TYPE ON ddname statement described in *Displaying Specific Messages: The TYPE Statement* on page 131 instead of the LOG facility.

You add a LOG statement for each file in which you are storing transactions. The syntax for the LOG statement is

```
LOG category [ON ddname] [MSG {ON|OFF}]
```

where:

category

Is the type of transaction to be logged. The types are:

TRANS are all transactions processed by the request.

ACCEPT are transactions accepted into the data source.

DUPL are transactions rejected because of an ON MATCH REJECT phrase (the transactions have field values that match those in the data source).

NOMATCH are transactions rejected because of an ON NOMATCH REJECT phrase (the transactions have field values that do not match values in the data source).

INVALID are transactions rejected because of data values that failed validation tests.

FORMAT are transactions rejected because of data values that have invalid formats (for example: a numeric field containing letters; an alphanumeric field with more characters than allowed by the format). Any non-CRTFORM transaction that fails an ACCEPT test can also be logged to this file.

ddname

The ddname of the file to which you are writing.

MSG

Controls the display of rejection messages (messages displayed on the terminal when a transaction is rejected). The default setting is ON, except for ACCEPT where the default is OFF. The ON setting enables the display of rejection messages.

You can log messages on six files in one request. If the files existed before the user executed the request, the logged transactions replace the file contents.

How the request stores transactions depends on the statement used to read them in.

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

When you allocate the files, you must assign each file a record length just large enough to hold the transaction. How you determine the length depends on how the request reads transactions:

FIXFORM and FREEFORM	Define the record length as the length of the longest logical transaction record, including blanks and commas between the fields. Remember that a logical transaction record can extend over more than one line in the transaction data source (but is recorded as one line in the log file).
PROMPT	Define the record length as the sum of the lengths of the fields as defined by the FORMAT attribute (for example, a field having a format of D12.2 has a length of 12), plus one byte for each field, plus one more byte.
CRTFORM	Define the record length as the sum of the lengths of the fields as defined by the FORMAT attribute (for example, a field having a format of D12.2 has a length of 12), plus one byte for each CRTFORM, plus one more byte.

The sample request below updates employee salaries and logs the transactions on five separate files. The original transaction data source was stored in file ddname SALFILE. Note the VALIDATE statement that determines whether the salary in each transaction exceeds $50,000.

```
MODIFY FILE EMPLOYEE

LOG TRANS    ON ALLTRANS
LOG ACCEPT   ON GOODTRAN
LOG NOMATCH  ON NOEMPL
LOG INVALID  ON BIGSAL
LOG FORMAT   ON BADFORM

PROMPT EMP_ID CURR_SAL
VALIDATE
  SAL_TEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

Note the five files specified in the LOG statements:

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

Syntax: How to Control the Printing of Rejection Messages

The MSG option on a LOG statement allows you to control the display of FOCUS automatic rejection messages. You can replace these messages by shutting them off and displaying your own messages using the TYPE command. The FOCUS messages are the following:

- For transactions rejected because of an ON MATCH REJECT phrase (the transactions have values that match values in the data source)

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

  where *n* is the transaction number and *segment* is the data source segment containing the data value that matched the transaction value.

- For transactions rejected because of an ON NOMATCH REJECT phrase (the transactions have values that do not match values in the data source)

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

  where *n* is the transaction number and *segment* is the data source segment containing the data field that failed to match the transaction value.

- For transactions rejected because of values that failed validation tests

  ```
  (FOC421)TRANS n REJECTED INVALID field
  ```

  where *n* is the transaction number and *field* is the return code field.

- For transactions read in using FIXFORM that were rejected because of values with format errors or ACCEPT errors

  ```
  (FOC428)TRANS n REJECTED FORMAT COL m FLD field
  ```

  where *n* is the transaction number, *m* is the first column of the field having the error, and *field* is the data field containing the error.

- For transactions read in using FREEFORM and PROMPT that were rejected because of values with format errors

  ```
  (FOC210) THE DATA VALUE HAS A FORMAT ERROR: v
  ```

  where *v* is the data value.

- For transactions read in using CRTFORM that were rejected because of values with format errors

  ```
  SCREEN REJECTED.. FORMAT ERROR IN FIELD field
  ```

where *field* is the data field with the format error.

- For transactions read in using CRTFORM or PROMPT that were rejected because a value failed in an ACCEPT test

  ```
  (FOC534) Data Value is not among the acceptable values for: field
  ```

  where *field* is the data field containing the error.

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

You may want to replace these messages with your own. To do so, use the TYPE statement described in *Displaying Specific Messages: The TYPE Statement* on page 131. To turn off the FOCUS messages, use the LOG statement with this syntax

```
LOG category [ON ddname] MSG {ON|OFF}
```

where:

category

Is the type of transaction that triggers the rejection message: DUPL, NOMATCH, INVALID, and FORMAT. These types are described previously in *How to Log Transactions in Sequential Files* on page 139.

ON *ddname*

Logs the transaction in a file defined by *ddname*. This option is described previously in *How to Log Transactions in Sequential Files* on page 139.

MSG

Is the parameter that turns FOCUS rejection messages ON (the default) or OFF.

For example, this request shuts off the automatic NOMATCH message and replaces it with another message:

```
MODIFY FILE EMPLOYEE
LOG NOMATCH MSG OFF
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH TYPE
     "THIS EMPLOYEE IS NOT RECORDED IN THE DATABASE"
     "DID YOU ENTER THE ID NUMBER CORRECTLY?"
     "THE NUMBER YOU ENTERED WAS: <EMP_ID"
  ON NOMATCH REJECT
DATA
```

Note that you may combine logging and the display of rejection messages in one LOG statement. For example, to both log transactions rejected because of the ON NOMATCH REJECT phrase and shut off the FOCUS message that results from those transactions, you can use this LOG statement:

```
LOG NOMATCH ON NOEMPL MSG OFF
```

Adding the logging facility enables the end user to deal with problem transactions after entering all the data.

Displaying Messages: The HELPMESSAGE Attribute

The HELPMESSAGE attribute enables you to specify a text message in the Master File of FOCUS data sources. The message is displayed in the TYPE area of MODIFY CRTFORMs.

Syntax: How to Specify a HELPMESSAGE Attribute

The syntax for specifying the HELPMESSAGE attribute in the Master File is

```
FIELDNAME=name, ALIAS=alias, FORMAT=format,
  HELPMESSAGE= text...,$
```

where:

text

Is a one-line text message up to 78 characters, which may include all characters and digits. Text containing a comma must be enclosed in single quotation marks; leading blanks are ignored.

For example:

```
FIELDNAME=LAST_NAME, ALIAS=LN, FORMAT=A10,
ACCEPT = SMITH JONES,
HELPMESSAGE = 'LAST_NAME MUST BE SMITH, OR JONES',$
```

The field for LAST_NAME has an ACCEPT attribute that tests values entered for that field. If a value other than Smith or Jones is entered, the following messages will be displayed:

```
(FOC534) DATA VALUE IS NOT AMONG ACCEPTABLE VALUES FOR LAST_NAME
LAST_NAME MUST BE SMITH, OR JONES
```

The HELPMESSAGE attribute can be used with a field that has an ACCEPT test (see the *Describing Data* manual), or any other field in the Master File.

Messages specified with the HELPMESSAGE attribute are displayed when:

- The value entered for a data source field is invalid according to the ACCEPT test for that field.

- The value entered for a data source field causes a format error.
- The user places the cursor in the data entry area for a particular field and presses a predefined PF key.

Regardless of the condition that triggers display of the message specified with the HELPMESSAGE attribute, the same message will appear.

Displaying Messages: Setting PF Keys to HELP

In order to see the HELPMESSAGE text for a field on the CRTFORM, set a PF key to HELP before executing the MODIFY procedure. To set a PF key, enter

```
SET PFnn = HELP
```

where:

nn

Is the number of the PF key you want to define as your HELP key.

To display a message for a particular field, position the cursor on the data entry area for that field on the CRTFORM and press the defined PF Key. If no message has been specified for the field, the following message will be displayed:

```
NO HELP AVAILABLE FOR THIS FIELD.
```

Case Logic

Case Logic allows you to branch to different parts of MODIFY requests during execution. This enables you to construct more complex MODIFY requests. For example, Case Logic requests can offer the terminal operator the choice of different procedures, process different transaction records differently, or update multiple segment instances with a single transaction.

Case Logic also extends the use of the NEXT statement to process segment chains and facilitates modifying multiple unique child segments.

To prepare a request using Case Logic, you divide the request into sections called cases. Each case is labeled, allowing you to branch to the case from elsewhere in the request.

Syntax: How to Use a Case Statement

Each case begins with the statement

```
CASE {AT START|casename}
```

where:

`AT START`

Indicates that the case is to be executed only at the beginning of the request. This case is called the START case.

`casename`

Is a label of up to 12 characters that does not contain embedded blanks or the characters:

`+ - * / & $ ' "`

Each case ends with the statement:

`ENDCASE`

The CASE and ENDCASE statements must both be on lines by themselves.

The first case in the request, the one immediately following the MODIFY command, needs neither a beginning nor an ending statement. It is automatically assigned the label TOP. Note, however, that if the request contains only one case, you may want to begin the case with the statement CASE TOP and end it with ENDCASE. This allows you to branch to the beginning of the request from its middle.

The following request updates employee salaries in the EMPLOYEE data source. If the salary is above $50,000, the request has the user retype the value to confirm it:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
IF CURR_SAL GT 50000 GOTO CONFIRM ELSE GOTO NEWSAL;

CASE NEWSAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE

CASE CONFIRM
TYPE
  "THE SALARY YOU ENTERED EXCEEDS $50,000"
  "PLEASE REENTER THE SALARY TO CONFIRM IT"
  "OR ENTER A NEW SALARY"
PROMPT CURR_SAL
GOTO NEWSAL
ENDCASE
DATA
```

This request consists of three cases: the TOP case, the NEWSAL case, and the CONFIRM case. (The blank lines between cases are there to enhance readability and are not required.)

The TOP case contains the first two statements in the request:

```
PROMPT EMP_ID CURR_SAL
IF CURR_SAL GT 50000 GOTO CONFIRM
```

The TOP case prompts you for an employee ID and new salary. It then tests the salary value you entered. If the salary is more than $50,000, it branches to the CONFIRM case. Otherwise, the request proceeds with the next case.

The next case is the NEWSAL case. This case updates the employee salaries. After the update, the request automatically returns to the beginning of the TOP case to prompt for the next employee ID and salary.

The third case is the CONFIRM case. This is where the request branches if you enter a salary higher than $50,000. The case asks you to reenter the salary. It then branches to the NEWSAL case to enter the salary into the data source.

This is the order of cases executed if you enter a salary lower than $50,000:

1. The TOP case.
2. The NEWSAL case.
3. Back to the TOP case.

This is the order of cases executed if you enter a salary higher than $50,000:

1. The TOP case.
2. The CONFIRM case.
3. The NEWSAL case.
4. Back to the TOP case.

Rules Governing Cases

The following rules apply to cases:

- Each case (except for the TOP case) must begin with a CASE statement and end with an ENDCASE statement; both statements must appear on separate lines.
- Each case must have a unique name within the MODIFY request.
- The TOP case is always the first case in the procedure. It has no beginning or ending case statements. No other case may be labeled TOP.
- If the TOP case has both CRTFORM and COMPUTE commands, the CRTFORM (data entry) is processed before the computation.

- There can be only one START case. If you include a START case, it must come after the TOP case. The START case is discussed in *Executing a Case at the Beginning of a Request Only: The START Case* on page 149.
- No case may be named EXIT. The label EXIT refers to the end of the request.
- Except for the TOP case, which must come first, and the START case, which follows after, the cases may appear in the request in any order.
- Except for the TOP and START cases, you can execute a case only by using a GOTO, PERFORM, or IF statement to branch to it.
- At the end of a case, the request branches back to the TOP case unless a GOTO or IF statement states otherwise.
- You cannot branch to the middle of a case, only to its beginning.

  Each case must contain complete MODIFY statements, not phrases or fragments. For example, the following case is illegal because ON NOMATCH REJECT is a phrase belonging to the MATCH statement.

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

- Cases cannot be nested; that is, you cannot put a case within another case. Each case must end before another can begin.
- You cannot have a statement between two cases except for comments. As soon as one case ends, the next case must begin.
- Certain MODIFY statements are global and apply to the request as a whole. We recommend that these statements follow the last case:

  ```
  START
  STOP
  LOG
  DATA
  CHECK
  ```

- Cases do not allow you to use either the FREEFORM or the PROMPT statement in requests with FIXFORM or CRTFORM statements. You also cannot use more than one FIXFORM statement with CRTFORMs. For using FIXFORM statements with CRTFORMs, see *Designing Screens With FIDEL* on page 227. You can mix FREEFORM statements with PROMPT statements in one request, and one FIXFORM statement with CRTFORM statements.
- There is no limit to the number of cases you can use in a MODIFY request.

- If a request repeatedly executes a case that has a CRTFORM, the case can produce up to 75 TYPE messages. If it produces more, FOCUS aborts the request.
- If you use fields with D. and T. prefixes in TYPE statements and CRTFORMs, a MATCH or NEXT statement must precede the fields, either in the same case or in a previously executed case (but not before the TOP case).

Executing a Case at the Beginning of a Request Only: The START Case

You can have your request begin execution with an initial case that is never executed afterwards. This case is called the START case and begins with the label:

```
CASE AT START
```

You cannot branch from other cases to the START case, but you can branch from the START case to other cases. If you do not branch to another case, the START case passes control to the TOP case. Note that the START case comes after the TOP case in the text of the request.

The following request counts how many employee salaries it updates. However, it starts counting from three:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH COMPUTE
     SALCOUNT/I4 = SALCOUNT + 1;
  ON MATCH UPDATE CURR_SAL
TYPE AT END
  "<SALCOUNT SALARIES PROCESSED"

CASE AT START
COMPUTE
  SALCOUNT = 3;
ENDCASE
DATA
```

The START case initializes the SALCOUNT counter to 3. After that, the request does not need to refer to the case again.

Note that temporary fields used in the START case that appear earlier in the request must have their formats defined there.

Branching to Different Cases: The GOTO, PERFORM, and IF Statements

Three statements branch to other cases:

- The GOTO statement, which branches unconditionally to another case. After the case executes, control returns to the TOP case.

- The PERFORM statement, which branches unconditionally to another case. When the case called by the PERFORM reaches ENDCASE, control returns to the statement following the PERFORM.
- The IF statement, which branches to GOTO or PERFORM as above, depending on the value of a logical expression.

Syntax: How to Branch to Another Case With GOTO

GOTO statements unconditionally branch to another case. The syntax is

```
GOTO location
```

where:

location
Is one of the following:

TOP branches to the beginning of the TOP case.

ENDCASE branches to the end of the case. If the case was called by a PERFORM statement either directly or indirectly (for example, a PERFORM statement called a case that branched to this case), then control returns to the statement after the most recently executed PERFORM statement. Otherwise, the request branches back to the TOP case.

casename branches to the beginning of the specified case.

variable branches to the beginning of the case whose name is the value of the temporary field *variable*. The temporary field must have a format of A12.

EXIT terminates the request. This is useful when you want to halt execution before the last transaction in a data source or the transaction specified by the STOP command. Note that the statement GOTO EXIT is legal even in MODIFY requests without cases.

If a case does not have a GOTO statement, it branches to the TOP case upon completion unless a PERFORM or IF statement branches somewhere else.

Syntax: How to Use a PERFORM Statement

The PERFORM statement causes the request to branch to another case, executes that case, then returns control to the statement after the most recently executed PERFORM statement. The syntax is

```
PERFORM  location
```

where:

location

Is one of the following:

TOP branches to the beginning of the TOP case. All return points are cleared and the procedure continues as if no PERFORM statement had executed.

ENDCASE branches to the end of the case. If the case was called by another PERFORM statement, either directly or indirectly (for example, a PERFORM statement called a case that branched to this case), then control returns to the statement after the most recently executed PERFORM statement. Otherwise, the request branches back to the TOP case.

casename branches to the beginning of a specified case.

variable branches to the beginning of the case whose name is the value of the temporary field *variable*. The temporary field must have a format of A12.

EXIT terminates the request.

A PERFORM statement can branch to a case containing a GOTO or IF statement that branches to a second case. The second case can branch to a third case, and so on until the request encounters an ENDCASE statement at the end of a case. Control then returns to the statement after the most recently executed PERFORM statement.

A PERFORM statement can branch to a case containing a PERFORM statement that leads to other cases. When the request encounters an ENDCASE statement at the end of a case, control returns to the statement after the most recently executed PERFORM statement. Control eventually returns to the original PERFORM.

If a case branches to the TOP case, control does not return to the last PERFORM. Rather, the request begins a new cycle starting from the TOP case. All PERFORM return points are cleared.

Example: Using the PERFORM Statement

This sample request updates employee salaries. If a user enters a salary greater than $50,000, the request checks the employee ID against a list of IDs in the sequential data source EMPLIST. If the employee is listed, the request updates the salary; otherwise, it asks the user to re-enter the information. The request is:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
PERFORM EMPCHECK
PERFORM UPSAL
TYPE
  "SALARY OF EMPLOYEE <EMP_ID UPDATED"
```

```
CASE EMPCHECK
IF CURR_SAL LE 50000 GOTO ENDCASE;
COMPUTE
  RAISE_OK/A3 = DECODE EMP_ID (EMPLIST ELSE 'NO');
IF RAISE_OK IS 'NO' THEN PERFORM TOP;
ENDCASE

CASE UPSAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
ENDCASE
DATA
```

Supposing the data source EMPLIST contained the following data:

```
071382660 YES
451123478 YES
```

A sample execution might go as follows:

1. The request prompts you for an employee ID and a salary. You enter ID 818692173 and a salary of $35,000.
2. The PERFORM EMPCHECK statement branches to the EMPCHECK case.
3. Since the salary is less than $50,000, the PERFORM ENDCASE phrase returns control to the statement after the PERFORM EMPCHECK statement (PERFORM UPSAL).
4. The PERFORM UPSAL statement branches to the UPSAL case.
5. The case updates the salary and passes control to the TYPE statement (the statement after the most recently executed PERFORM statement).
6. The TYPE statement displays the message:

   ```
   SALARY FOR EMPLOYEE 8188692173 UPDATED
   ```

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

13. The PERFORM UPSAL statement branches to the UPSAL case, which updated the salary. Control then passes to the TYPE statement (the statement after the most recently executed PERFORM statement).
14. The TYPE statement displays a message:

```
SALARY FOR EMPLOYEE 071382660 UPDATED
```

15. Control goes to the beginning of the TOP case.

Reference: Rules for PERFORM Statements

- PERFORM statements can be nested; that is, one PERFORM statement can call a case containing a second PERFORM statement. PERFORM statements can be nested to any depth, limited only by available memory. If memory runs out, FOCUS displays the message:

  ```
  (FOC187) PERFORMS NESTED TOO DEEPLY
  ```

- REPEAT statements can contain PERFORM statements. When control returns to the statement after the most recently executed PERFORM statement, the REPEAT statement resumes execution. For example:

  ```
  REPEAT 5 TIMES
    PERFORM ANALYSIS
    COMPUTE AMOUNT/D8.2 = RECEIPTS + AWARDS;
  ENDREPEAT
  ```

  Each pass of this REPEAT statement executes the ANALYSIS case, then computes the value of the AMOUNT field.

- When a PERFORM statement branches to a case, you can return control to the PERFORM before the end of the case by including the GOTO ENDCASE or PERFORM ENDCASE statement in the case.

Syntax: How to Branch to Another Case With IF

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

```
IF expr [THEN] {GOTO|PERFORM} location1 [ELSE {GOTO|PERFORM} location2]
```

where:

expr

Is any logical expression legal in a DEFINE or COMPUTE IF statement (see the *Creating Reports* manual). For example:

```
IF CURR_SAL GT 50000
IF SALARY/12 LT GROSS
IF LAST_NAME CONTAINS 'BLACK'
IF (CURR_SAL GT SALARY) OR
     (CURR_JOB CONTAINS 'B')
```

Note that literals must be enclosed in single quotation marks. Parentheses are necessary if the expression is compound.

IF expressions cannot compare data source fields unless they are used in or following MATCH or NEXT statements (see *Branching to Different Cases: The GOTO, PERFORM, and IF Statements* on page 149).

location1, location2

The options are:

TOP branches to the TOP case.

ENDCASE branches to the end of the case (the request then branches to the TOP case or to the statement after the most recently executed PERFORM statement).

case1 branches to the case named *case1*.

var branches to the case whose name is contained in the temporary field *var*.

EXIT terminates the request.

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

An IF statement can extend over several lines, but must end with a semicolon (;).

Like IF statements in TABLE requests and Dialogue Manager control statements, Case Logic IF statements can be nested. You can nest IF statements so that if the outer IF expression is true, the inner IF is executed. Place the inner IF phrase within parentheses following the THEN phrase.

Example: **IF Statement**

```
IF expression1
THEN (IF expression2
THEN (IF expression3 GOTO case4 ELSE GOTO case3)
ELSE GOTO case2)
ELSE GOTO case1;
```

You can also nest IF statements so that if the outer IF expression is false, the inner IF is executed. You place the inner IF statement after the ELSE phrase. The inner IF does not need parentheses:

```
     IF expression1 THEN GOTO case1
ELSE IF expression2 THEN GOTO case2
ELSE IF expression3 THEN GOTO case3
ELSE...;
```

The following request offers the user a choice between deleting a segment instance and including a new one:

```
MODIFY FILE EMPLOYEE
COMPUTE CHOICE/A6=;
TYPE
  "ENTER 'UPDATE' TO UPDATE A SALARY"
  "ENTER 'DELETE' TO DELETE AN EMPLOYEE"
PROMPT CHOICE


      IF CHOICE IS 'UPDATE' THEN GOTO UPDSEG
ELSE  IF CHOICE IS 'DELETE' THEN GOTO DELSEG
ELSE GOTO TOP;


CASE UPDSEG
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE


CASE DELSEG
PROMPT EMP_ID
MATCH EMP_ID
  ON MATCH DELETE
  ON NOMATCH REJECT
ENDCASE
DATA
```

This request has three cases:

- The TOP case defines a variable called CHOICE, which will contain your response to its menu:

  If you enter UPDATE, the request branches to the UPDSEG case.

  If you enter DELETE, the request branches to the DELSEG case.

  If you enter neither, it reprompts you for another response by branching back to the beginning of the case.

- The UPDSEG case prompts you for the employee ID and new salary, and updates the employee's salary.

- The DELSEG case prompts you for the employee ID, and deletes that ID from the data source.

Rules Governing Branching

The following rules govern the sequence of case execution and branching:

- The request first executes the START case, if there is one. It then executes the TOP case, unless the START case branches to another case.
- If a case does not execute a GOTO statement, a PERFORM statement, or an IF statement to branch to another case, it branches to the TOP case by default. This is true of both the START and TOP cases. However, if the case was called by a PERFORM statement either directly or indirectly (for example, a PERFORM statement called a case that branched to a case that branched to this case), then control returns to the statement after the most recently executed PERFORM statement.
- A case can branch to itself.
- Branching to the TOP case, whether by a GOTO TOP statement, PERFORM TOP statement or by default, deactivates all data fields (field activation and deactivation are described in *Active and Inactive Fields* on page 204) and increments the transaction counter by one.
- When you branch to a case, you always branch to the beginning of the case. You can never branch into the middle of a case.
- If one case contains a MATCH or NEXT statement that selects a particular segment instance, it can branch to another case that modifies the child segment chain belonging to the same instance. The second case need not reselect the parent instance, but it must contain at least one MATCH statement. For example, the segment EMPINFO (key field EMP_ID) has the child segment SALINFO (key field PAY_DATE). You can include a new SALINFO segment with this request:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO NEWPAY


CASE NEWPAY
MATCH PAY_DATE
  ON NOMATCH INCLUDE
  ON MATCH REJECT
ENDCASE
DATA
```

The second case, NEWPAY, modifies the segment chain descended from the segment instance selected in the TOP case.

GOTO, PERFORM, and IF Phrases in MATCH Statements

You can use GOTO, PERFORM, and IF statements in MATCH and NEXT statements, where they form part of ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrases. IF phrases in MATCH and NEXT statements can use data source fields in expressions. To do this, affix the D. prefix to the field name. For example, the phrase

```
ON MATCH IF CURR_SAL LT D.CURR_SAL ...
```

tests whether the incoming value of CURR_SAL is less than the data source value of CURR_SAL. The data source value must either be in the segment instance that the MATCH or NEXT statement is processing or in a parent instance along the segment path (the parent, the parent's parent, and so on, up to the root segment).

For example, this request does not accept a new salary for an employee if it is less than the employee's present salary:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH IF CURR_SAL LT D.CURR_SAL GOTO ERROR;
  ON MATCH UPDATE CURR_SAL

CASE ERROR
TYPE
  "YOU ENTERED A NEW SALARY"
  "LESS THAN THE EMPLOYEE'S PRESENT SALARY"
  "PLEASE REENTER DATA"
ENDCASE
DATA
```

This request consists of two cases:

- The TOP case prompts you for an employee ID and new salary. If the employee ID is in the data source, the case tests whether the new salary is less than the present one. If the new salary is lower, it branches to the ERROR case. Otherwise, it updates the salary and branches back to the TOP case.
- The ERROR case warns you that the salary you entered is unacceptable and branches back to the TOP case.

If the MATCH statement specifies fields in multiple segments (the technique of matching across segments, described in *Modifying Segments in FOCUS Structures* on page 87), the GOTO, PERFORM and IF phrases in the statement are only executed when the MATCH statement modifies the last segment. For example, this request adds instances to the EMPINFO, SALINFO, and DEDUCT segments:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID PAY_DATE DED_CODE
GOTO ADD


CASE ADD
MATCH EMP_ID PAY_DATE DED_CODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
ENDCASE


CASE MESSAGE
TYPE
  "NEW INSTANCE ADDED"
ENDCASE
DATA
```

The ADD case branches to the MESSAGE case only when it includes a new instance in the segment containing the DED_CODE field. If you want the case to branch to the MESSAGE case when it includes a new instance in any of the segments, then write the case with a separate MATCH statement for each segment it searches:

```
CASE ADD
MATCH EMP_ID
  ON MATCH CONTINUE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
MATCH PAY_DATE
  ON MATCH CONTINUE
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
MATCH DED_CODE
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO MESSAGE
ENDCASE
```

Example: Using Case Logic and Validation Tests

You can also branch to other cases when an incoming field value fails a validation test. Do this by including GOTO, PERFORM, and IF statements as part of the ON INVALID phrase. For example, this request processes transactions with salaries higher than $50,000 in a separate case:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL


CASE NEWSAL
PROMPT CURR_SAL
VALIDATE
  SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
  ON INVALID GOTO HIGHSAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE


CASE HIGHSAL
TYPE
  "SALARY ABOVE $50,000 NOT ALLOWED"
  "RETYPE SALARY BELOW"
GOTO NEWSAL
ENDCASE
DATA
```

Case Logic Applications

This section discusses some examples of applications for Case Logic that extend the capabilities of MODIFY requests. The applications are:

- Looping through segment chains using the NEXT statement.
- Modifying multiple unique segments.
- Using Case Logic to offer user choices.
- Using Case Logic to process transaction data sources.
- Using Case Logic to process transactions based on the values of their fields.
- Using Case Logic to process transactions with bad values.

Syntax: How to Loop Through a Segment Chain With the NEXT Statement

The NEXT statement, discussed in *Selecting the Instance After the Current Position: The NEXT Statement* on page 102, modifies or displays the next segment instance after the current position in the data source. Using Case Logic, you can use NEXT statements to process entire segment chains.

For an entire segment chain to be displayed, the request must branch back to the beginning of the NEXT statement. Put the NEXT statement in a separate case, as shown below:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH TYPE
     "WAGES PAID TO EMPLOYEE #<EMP_ID"
  ON MATCH GOTO SALHIST

CASE SALHIST
NEXT DAT_INC
  ON NEXT TYPE "<D.DAT_INC <D.SALARY"
  ON NEXT GOTO SALHIST
  ON NONEXT GOTO TOP
ENDCASE
DATA
```

This request consists of two cases:

- The TOP case prompts you for an employee ID and branches to the SALHIST case.
- The SALHIST case contains one NEXT statement that displays the next instance of the employee's salary chain. The case then branches back to the its beginning to display the next instance. When it reaches the end of the chain, it branches back to the TOP case.

To return to the beginning of a segment chain, use the REPOSITION statement. The syntax is

```
REPOSITION field
```

where *field* is any field of the segment. The REPOSITION statement allows you to return to the beginning of the segment chain you are now modifying, or to the beginning of the chain of any of the parent instances along the segment path (that is, the parent instance, the parent's parent, and so on to the root segment). You can then search the segment chain from the beginning.

The following request allows you to allocate a new monthly pay for a selected employee for each pay date. The request accumulates each pay in a total. If this total pay exceeds the employee's yearly salary, the request returns to the first pay date to permit you to enter new values for the entire chain:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO PAYLOOP
CASE PAYLOOP
NEXT PAY_DATE
  ON NONEXT GOTO TOP
  ON NEXT TYPE
     "EMPLOYEE ID: <EMP_ID"
     "PAY DATE: <D.PAY_DATE MONTHLY PAY: <D.GROSS"
  ON NEXT PROMPT GROSS.ENTER MONTHLY PAY:.
  ON NEXT COMPUTE
     TOTAL_PAY/D10.2 = TOTAL_PAY + GROSS;
  ON NEXT IF TOTAL_PAY GT D.CURR_SAL GOTO ERROR;
  ON NEXT UPDATE GROSS
  ON NEXT GOTO PAYLOOP
ENDCASE

CASE ERROR
TYPE
  "TOTAL MONTHLY PAY EXCEEDS YEARLY SALARY"
  "REENTER PROPOSED PAY STARTING FROM"
  "THE FIRST PAY DATE"
REPOSITION PAY_DATE
COMPUTE TOTAL_PAY = 0;
GOTO PAYLOOP
ENDCASE
DATA
```

Note that the ERROR case in the example warns you that the sum of the figures you entered exceeds the employee's yearly salary. It then repositions the current position of the PAY_DATE field at the beginning of the segment chain and branches back to the PAYLOOP case, allowing you to reenter pay figures for the entire chain.

When you use INCLUDE, UPDATE, and DELETE actions in looping NEXT statements, note the following:

- Use the ON NEXT INCLUDE and ON NONEXT INCLUDE phrases only to add instances to segments of type S0 or blank. If you use these phrases to modify other segments, you will duplicate what is already there. The difference between the two phrases is:

  ON NEXT INCLUDE adds a new segment instance after the current position.

  ON NONEXT INCLUDE adds a new instance at the end of the segment chain.

- Use the ON NEXT UPDATE phrase without restriction. The phrase updates the segment instance at the current position. If you are looping with the NEXT statement, the phrase updates the entire chain.

- Use the ON NEXT DELETE phrase to delete entire segment chains. This phrase deletes the segment instance at the current position. If you are looping with the NEXT statement, the phrase deletes the entire chain, but only if you start at the beginning of a chain. Otherwise, the phrase deletes every second instance.

Note that the phrases ON NONEXT UPDATE and ON NONEXT DELETE are illegal and will generate error messages.

Example: Modifying Multiple Unique Segments

Modifying unique segments is described in *Modifying Segments in FOCUS Structures* on page 87. This section describes how to modify several unique segments descended from one parent using the CONTINUE TO method.

To modify multiple unique segments, prepare separate cases containing a MATCH or NEXT statement for each segment you are modifying. The sample request below illustrates this. The request loads data into the SUBSCRIBE data source, which records magazine subscribers, their mailing addresses, and expiration dates. The Master File is:

```
FILE=SUBSCRIB ,SUFFIX=FOC,$
SEGMENT=SUBSEG ,$
 FIELD=SUBSCRIBER      ,ALIAS=NAME    ,FORMAT=A35     ,$
SEGMENT=ADDRSEG,SEGTYPE=U,PARENT=SUBSEG               ,$
 FIELD=ADDRESS    ,ALIAS=ADDR  ,FORMAT=A40            ,$
SEGMENT=EXPRSEG,SEGTYPE=U,PARENT=SUBSEG               ,$
 FIELD=EXPR_DATE       ,ALIAS=EXDATE  ,FORMAT=I6DMYT  ,$
```

The following MODIFY request loads the data:

```
MODIFY FILE SUBSCRIB
PROMPT SUBSCRIBER
MATCH SUBSCRIBER
  ON NOMATCH INCLUDE
  ON MATCH CONTINUE
GOTO NEWADDR

CASE NEWADDR
PROMPT ADDRESS
MATCH SUBSCRIBER
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO ADDRESS
     ON MATCH REJECT
     ON MATCH GOTO NEWDATE
     ON NOMATCH INCLUDE
     ON NOMATCH GOTO NEWDATE
ENDCASE

CASE NEWDATE
PROMPT EXPR_DATE
MATCH SUBSCRIBER
  ON NOMATCH REJECT
  ON MATCH CONTINUE TO EXPR_DATE
     ON MATCH REJECT
     ON NOMATCH INCLUDE
ENDCASE
DATA
```

Note the last two cases in the request:

- The NEWADDR case loads subscriber addresses into the unique segment ADDRSEG. The case examines the ADDRSEG segment. Does the subscriber have a mailing address listed? If not, the request includes the new address. In either event, the request continues to the NEWDATE case.
- The NEWDATE case loads expiration dates into the sibling unique segment EXPRSEG. It examines the EXPRSEG segment with the EXPR_DATE field. Does the subscriber have a magazine expiration date? If not, the request includes the new expiration date. If the subscriber has an expiration date, the request checks to determine whether it gave the subscriber a new address.

  If the request gave the subscriber a new address, the request does not reject the transaction.

  If the request did not give the subscriber a new address, the request rejects the transaction.

If you were to include the MATCH statements in one case, the request would reject a transaction if the subscriber already had either an address or an expiration date. Since you want the transaction rejected only if the subscriber already has both, separate the MATCH statements into separate cases.

Procedure: How to Use Case Logic to Offer User Selections

You can use Case Logic to offer users a selection of options. The request below offers a choice between updating employee salaries, monthly pay, or addresses:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO MENU

CASE MENU
TYPE
"TO UPDATE THE EMPLOYEE'S SALARY, TYPE 'SALARY' "
"TO UPDATE THE EMPLOYEE'S MONTHLY PAY, TYPE 'PAY' "
"TO UPDATE THE EMPLOYEE'S ADDRESS, TYPE 'ADDRESS' "
COMPUTE CHOICE/A7=;
PROMPT CHOICE
       IF CHOICE IS 'SALARY' THEN GOTO SALARY
 ELSE  IF CHOICE IS 'PAY'THEN GOTO PAY
 ELSE  IF CHOICE IS 'ADDRESS'THEN GOTO ADDRESS;
TYPE "ILLEGAL CHOICE, PLEASE TYPE ENTRY AGAIN"
GOTO MENU
ENDCASE
CASE SALARY
PROMPT CURR_SAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
ENDCASE
CASE PAY
PROMPT PAY_DATE GROSS
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON MATCH UPDATE GROSS
ENDCASE
CASE ADDRESS
PROMPT TYPE ADDRESS_LN1 ADDRESS_LN2
MATCH TYPE
  ON NOMATCH REJECT
  ON MATCH UPDATE ADDRESS_LN1 ADDRESS_LN2
ENDCASE
DATA
```

Procedure: How to Use Case Logic to Process Transaction Data Sources

You can use Case Logic to process records in a transaction data source in different ways. For example, each transaction record contains a field that defines what type of record it is. The MODIFY request can use these record types to branch to the appropriate case and process the transaction.

The following request processes two record types: type A updates employee department assignments and job codes; type B updates salaries and classroom hours. The record type field (called RTYPE) is the last field in each record. It contains either the letter A or B, depending on the record type.

```
MODIFY FILE EMPLOYEE
COMPUTE RTYPE/A1=;
FIXFORM  X26 RTYPE/1
      IF RTYPE IS 'A' THEN GOTO TYPE_A
 ELSE IF RTYPE IS 'B'THEN GOTO TYPE_B;
TYPE "BAD RECTYPE VALUE"
GOTO TOP

CASE TYPE_A
FIXFORM X-27 EMP_ID/9 X1 DEPARTMENT/10
FIXFORM X1 CURR_JOBCODE/3 X3
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE DEPARTMENT CURR_JOBCODE
ENDCASE

CASE TYPE_B
FIXFORM X-27 EMP_ID/9 X1 CURR_SAL/8 X1 ED_HRS/6 X2
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL ED_HRS
ENDCASE
DATA ON FIXTYPE
END
```

Notice the three FIXFORM statements: one in each of the cases. Only the statement in the TOP case reads a record from disk or tape. The other two statements redefine the record for the case.

Also note that each of these two statements begins with X-27, which allows the case to redefine the 27-byte record from the beginning. Always place the notation X-*n* at the beginning of the FIXFORM statement that is redefining the record, not at the end of the previous FIXFORM statement.

A FIXFORM statement reads a new record from disk or tape if one of these conditions are met:

- The statement is the first FIXFORM statement in the request.

- The statement defines records to be longer than they were defined before. For instance, if one FIXFORM statement defines a record of 80 bytes, and the next FIXFORM statement defines a record from the same data source as being 90 bytes, the second FIXFORM statement reads a new record.
- The statement reads records from a different data source than the one read previously. This is possible if the statement has the form

  ```
  FIXFORM ON ddname
  ```

  where *ddname* is the ddname of the second transaction data source. If the next FIXFORM statement does not have the ON ddname option, it too reads another record.

Procedure: How to Use Case Logic to Process Transactions Based on the Values of Their Fields

You can use Case Logic to process transactions depending on their field values. The following request updates employee salaries. If the user enters a salary higher than $50,000, the request checks the employee ID against a list of employees authorized for large salaries:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID
GOTO NEWSAL

CASE NEWSAL
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH PROMPT CURR_SAL
  ON MATCH IF CURR_SAL GT 50000 THEN GOTO HIGHSAL;
  ON MATCH UPDATE CURR_SAL
ENDCASE

CASE HIGHSAL
COMPUTE
  SALTEST = DECODE EMP_ID (HIGHPAY);
IF SALTEST NE 1 THEN GOTO WRONGSAL;
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH UPDATE CURR_SAL
ENDCASE

CASE WRONGSAL
TYPE
  "EMPLOYEE NOT AUTHORIZED FOR SALARY INCREASE"
  "PLEASE REENTER THE DATA"
ENDCASE
DATA
```

Procedure: How to Use Case Logic to Process Transactions With Bad Values

You can use Case Logic to process transactions with values that would otherwise cause the transactions to be rejected. You do this by combining GOTO and IF phrases with:

- The ON MATCH phrase, if you are adding new segment instances.
- The ON NOMATCH phrase, if you are updating or deleting instances.
- The ON INVALID phrase, if you are validating incoming data fields.

This request updates employee salaries. If it cannot find an employee record, it queries the user whether to include the transaction as a new employee record:

```
MODIFY FILE EMPLOYEE
PROMPT EMP_ID CURR_SAL
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH GOTO QUERY

CASE QUERY
COMPUTE CHOICE/A1=;
TYPE
  "EMPLOYEE ID NOT FOUND IN THE DATABASE"
  "INCLUDE THE TRANSACTION ANYWAY (Y/N)?"
PROMPT CHOICE
      IF CHOICE IS 'Y' THEN GOTO INCLUDE
 ELSE IF CHOICE IS 'N'THEN GOTO REJECT;
TYPE "PLEASE TYPE EITHER Y OR N"
GOTO QUERY
ENDCASE

CASE INCLUDE
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
ENDCASE

CASE REJECT
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH REJECT
ENDCASE
DATA
```

Tracing Case Logic: The TRACE Facility

The TRACE facility displays the name of each case that is entered during the execution of a MODIFY request. This is a useful tool for debugging large Case Logic requests.

You can allocate the output to a file or to your terminal. Then, add the word TRACE to the end of the MODIFY command line

```
MODIFY FILE filename TRACE
```

where:

filename

is the name of the FOCUS data source you are modifying.

When the TRACE facility is on, it lists in the HLIPRINT file the name of the case about to run

```
TRACE ===> AT CASE case
```

where:

case

Is the name of the case.

Note that if you are using FIDEL and displaying the TRACE output on the terminal, the following happens. When you enter a CRTFORM screen, the screen clears and displays the name of the next case. Clear the screen, and the next CRTFORM screen appears.

The request and sample execution below illustrate the use of the TRACE facility:

```
MODIFY FILE EMPLOYEE TRACE
PROMPT EMP_ID CURR_SAL
IF CURR_SAL GT 50000 GOTO HIGHSAL
ELSE GOTO UPDATE;

CASE UPDATE
MATCH EMP_ID
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
ENDCASE

CASE HIGHSAL
TYPE
  " "
  "YOU ENTERED A SALARY ABOVE $50,000"
  " "
PROMPT CURR_SAL.PLEASE REENTER THE SALARY.
IF CURR_SAL GT 50000 GOTO HIGHSAL
ELSE GOTO UPDATE;
ENDCASE
DATA
```

The following is a sample execution of the previous request:

```
> EMPLOYEE     ON 10/04/98 AT 14.02.33
**** START OF TRACE ****
TRACE ===> AT CASE TOP
DATA FOR TRANSACTION  1

EMP_ID  = > 112847612
CURR_SAL  = > 67000
TRACE ===> AT CASE HIGHSAL

YOU ENTERED A SALARY ABOVE $50,000

PLEASE REENTER THE SALARY > 27000
TRACE ===> AT CASE UPDATE
TRACE ===> AT CASE TOP
DATA FOR TRANSACTION 2

EMP_ID = 0
```

Multiple Record Processing

Multiple record processing enables you to process multiple segment instances at one time. One important application is the use of multiple record processing with the FIDEL facility to enable the terminal operator to add, update, or delete several segment instances on one screen. This section discusses multiple record processing based on this application. However, you can apply the principles stated here to other applications as well.

Usually, a MODIFY request using FIDEL prompts you for a key field value, then uses the value to retrieve one segment instance. After you modify the instance, you enter the key field value to retrieve the next instance. This way, you modify segment instances one at a time.

Multiple record processing causes the request to retrieve multiple segment instances before FIDEL displays instance values. Each time the request retrieves an instance, it stores the instance values in a work area in memory called the Scratch Pad Area. The request continues to retrieve instances until it reaches a specified number.

After the request has retrieved the instances, FIDEL reads the instance values from the Scratch Pad Area and displays them all on one screen. The user can update these values and transmit the updated values back to the data source with one press of the Enter key.

Note: Text fields cannot be put into the Scratch Pad (HOLD).

You may also design a request that adds several instances at one time, or a request that both updates existing instances and adds new ones all on the same screen.

The REPEAT Method on page 170 describes multiple record processing using the REPEAT statement. This method requires only that you know the fields you want to process. However, it only enables you to process instances from one segment at a time.

Manual Methods on page 180 discusses manual methods that require you to know how instances are stored in the Scratch Pad Area. These methods are more powerful and enable you to process multiple segments at one time.

The REPEAT Method

One REPEAT statement collects segment instances and loads them into the Scratch Pad Area; another REPEAT statement retrieves the instances from the Area and uses them to modify the data source. This method does not require you to know how the instances are stored in the Area; however, you must process the instances sequentially, and you can process only one segment at one time.

Multiple record processing has four phases. They are:

1. **Selection.** The request selects the parent instance of the instances to be processed.
2. **Collection.** The request retrieves multiple segment instances and stores their data values in the Scratch Pad Area.
3. **Display.** The FIDEL facility displays the data on one screen for editing.
4. **Modification.** The request uses the edited data values to modify the data source.

The Selection Phase: Selecting the Parent Instance

To modify multiple instances in a segment, you must first identify the parent instance. (If you are modifying the root segment, skip this phase and start with *The Collection Phase: Storing Instances in a Buffer* on page 171.) Do this as you would any other request.

For example, the beginning of this request identifies an employee ID in the EMPLOYEE data source, allowing you to modify the employee's child segment instances:

```
MODIFY FILE EMPLOYEE
CRTFORM LINE 2
"************************************"
"* MONTHLY PAY UPDATE                *"
"************************************"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO COLLECT
```

If you are using multiple record processing only to create new instances, skip the collection phase and proceed directly to the display phase. The following MATCH statement adds a new employee ID to the data source. It then branches to the case NEWADDRESS where the display phase prompts the user for all the employees' addresses:

```
MODIFY FILE EMPLOYEE
CRTFORM
"ENTER EMPLOYEE'S ID: <EMP_ID"
MATCH EMP_ID
  ON MATCH REJECT
  ON NOMATCH INCLUDE
  ON NOMATCH GOTO NEWADDRESS
```

The Collection Phase: Storing Instances in a Buffer

During the collection phase, the request retrieves multiple segment instances and stores their values in the Scratch Pad Area.

After identifying the parent instance, read the child instances into the Scratch Pad Area (if you are modifying the root segment, reading the instances into the Area is your first step). You do this using the REPEAT statement, which the request executes repeatedly. Each time the request executes a REPEAT statement, the phrases in the statement retrieve one segment instance and store its data values in the Area.

Syntax: How to Use a REPEAT Statement

The syntax of the REPEAT statement is

```
REPEAT {*|count}[TIMES] [MAX limit] [NOHOLD]
   .
   .
phrases
   .
   .
ENDREPEAT
```

where:

count

Is an integer or temporary integer field determining the number of times the request executes the REPEAT. This value can be between 0 and 32,767, but should be no smaller than the number of segment instances you want to display on the FIDEL screen.

If this value is 0, the request does not execute the REPEAT (this allows you to skip a REPEAT if you are using a temporary field for this parameter). If the value is an asterisk, the REPEAT is executed 65,535 times. Once the REPEAT begins execution, the value cannot be changed.

TIMES

Is an optional word, which you can add to enhance readability.

limit

Is an integer specifying the maximum number of times the request can execute the REPEAT. Specify this parameter only if you are using a temporary field for the *count* parameter.

NOHOLD

Is an option that enables you to use REPEAT as a simple loop that executes any group of MODIFY statements repeatedly.

phrases

Are the MODIFY statements to be executed within the REPEAT statement. Each phrase must begin on a new line.

ENDREPEAT

Ends the statement. This phrase must be on a line by itself.

There are three types of REPEAT statements:

- Stacking REPEAT statements. These statements contain HOLD phrases that stack data into the Scratch Pad Area. They appear in the collection phase of multiple record processing.
- Retrieving REPEAT statements. These statements retrieve data placed in the Scratch Pad Area by the stacking REPEAT statements. They usually appear in the modification phase and in validation routines in multiple record processing.
- Simple REPEAT statements. These statements consist of any combination of MODIFY statements to be executed repeatedly. You indicate a simple repeat statement by specifying the NOHOLD option in the REPEAT phrase. Simple REPEAT statements neither stack data nor retrieve data from the Scratch Pad Area.

FOCUS determines the type of REPEAT statement in the following manner:

- If the statement specifies the NOHOLD option, it is a simple REPEAT statement.
- If the statement contains a HOLD phrase, it is a stacking REPEAT statement.
- If the statement neither specifies the NOHOLD option nor contains a HOLD phrase, it is a retrieving REPEAT statement.

The REPEAT statement can stand by itself, or it can be part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. For example:

```
REPEAT 12 TIMES

ON MATCH REPEAT 6
```

```
ON NEXT REPEAT BUFCOUNT MAX 10
```

Note: You cannot nest REPEAT statements; one statement must end before another can begin.

Two GOTO phrases especially apply to REPEAT statements. They are:

- GOTO ENDREPEAT. This phrase branches processing to the end of the REPEAT statement, increments the counter by 1, and executes the request REPEAT again.
- GOTO EXITREPEAT. This phrase branches processing to the first executable statement following the REPEAT loop.

This REPEAT saves the first five pay dates and monthly pay amounts in the EMPLOYEE data source in the Scratch Pad Area:

```
CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
    ON NEXT HOLD PAY_DATE GROSS
    ON NONEXT GOTO EXITREPEAT
ENDREPEAT
GOTO DISPLAY
ENDCASE
```

Note the ON NONEXT GOTO EXITREPEAT phrase. This specifies that if there are less than five employee IDs in the segment chain, the request will branch to the next statement after the REPEAT. If the ON NONEXT phrase was not included, the request would automatically branch back to the beginning of the request.

Syntax:

How to Store Instances With the HOLD Phrase

The REPEAT statement retrieves instances using MATCH and NEXT statements. Each time the REPEAT retrieves an instance, you may store the instance values in the Scratch Pad Area. Do this with the phrase

```
HOLD [SEG.]field-1 field-2 ... field-n
```

where *field-1* through *field-n* are the data fields whose values you want to save in the Scratch Pad Area. The specified fields can be data source fields or temporary fields. The data source fields must exist either in the instance or in a parent instance along the segment path (the parent of the instance, the parent's parent, and so on to the root segment). For example, the phrase

```
HOLD EMP_ID FIRST_NAME LAST_NAME CURR_SAL
```

stores the employee IDs, first and last names, and salaries of each retrieved instance in the Scratch Pad Area.

If you want to save the values of all the data fields in the instance, specify just one field with the SEG. prefix affixed to the front of the field name.

HOLD stores the fields whether they are active or inactive. To ensure that the fields placed in the Scratch Pad Area are active, use the ACTIVATE phrase described in *Active and Inactive Fields* on page 204.

The HOLD phrase can stand by itself, or it can be part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. If you use HOLD in ON NOMATCH and ON NONEXT phrases, you may specify only temporary fields and fields in parent instances along the segment path. If the list of fields is too long to fit on one line, repeat the word HOLD for each line you need. Some examples are:

```
HOLD EMP_ID LAST_NAME FIRST_NAME DEPARTMENT
HOLD CURR_JOBCODE ED_HRS

ON MATCH HOLD EMP_ID DEPARTMENT CURR_SAL

ON NONEXT HOLD DEPCODE
```

When a REPEAT statement containing a HOLD phrase begins execution, FOCUS clears the Scratch Pad Area of data stored from previous REPEATs.

The following is a piece of a MODIFY request that executes the collection phase:

```
CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
    ON NEXT HOLD PAY_DATE GROSS
    ON NONEXT GOTO DISPLAY
ENDREPEAT
GOTO DISPLAY
ENDCASE
```

Reference: The REPEATCOUNT and HOLDCOUNT Variables

Two variables assume values during the collection phase. These are:

- The REPEATCOUNT variable. This variable contains the value of the REPEAT counter.
- The HOLDCOUNT variable. This variable contains the current number of instances stored in the Scratch Pad Area.

If you design your request with Case Logic, you can test and branch on these variables. The following IF statement branches to the TOP case if the preceding REPEAT did not retrieve any segment instances:

```
IF HOLDCOUNT EQ 0 GOTO TOP
```

Please note the following values that the REPEATCOUNT and HOLDCOUNT variables take under these circumstances:

- When a REPEAT statement begins execution, REPEATCOUNT is set to 1.
- If a REPEAT is set to execute 0 times, REPEATCOUNT is set to 0.
- If the REPEAT beginning execution contains HOLD phrases, the Scratch Pad Area is cleared and HOLDCOUNT is set to 0. If the REPEAT does not contain HOLD phrases, HOLDCOUNT is unchanged.
- At each repetition of the REPEAT, REPEATCOUNT is increased by one. After each HOLD phrase is executed, HOLDCOUNT is increased by one.
- The REPEATCOUNT variable maintains its value after the REPEAT completes execution until the next REPEAT, even if the request branched from the REPEAT with a GOTO phrase.

  Note: A CRTFORM displaying records in the Scratch Pad Area can change the HOLDCOUNT value. For this reason, you may want to store the HOLDCOUNT value in a temporary field for use later in the request. For example, this COMPUTE statement saves the value of the HOLDCOUNT field in the temporary field BUFFNUMBER:

  ```
  COMPUTE BUFFNUMBER/I5 = HOLDCOUNT;
  ```

The Display Phase: Displaying Instances in One CRTFORM

After the request stores the segment instance values in the Scratch Pad Area, you display the values on one screen using the FIDEL facility (see *Designing Screens With FIDEL* on page 227). Since you use the same field names for all instances (multiple record processing can only modify one segment at a time), you must distinguish between instances. To do this, add subscripts to the fields using the form.

```
field(n)
```

where *n* (the subscript) is an integer greater than 0. The subscript indicates the instance that a field belongs to in the order that the instances are read from the Scratch Pad Area.

For example, this CRTFORM displays the employee IDs, departments, and salaries of five segment instances numbered 1 through 5:

```
CASE DISPLAY
IF HOLDCOUNT EQ 0 GOTO TOP;
COMPUTE
  BUFFNUMBER/I5=HOLDCOUNT;
CRTFORM LINE 9
   " MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
   " "
   " PAY DATE         AMOUNT PAID"
   " --------         -----------"
   "<D.PAY_DATE(1)    <T.GROSS(1)>"
   "<D.PAY_DATE(2)    <T.GROSS(2)>"
   "<D.PAY_DATE(3)    <T.GROSS(3)>"
   "<D.PAY_DATE(4)    <T.GROSS(4)>"
   "<D.PAY_DATE(5)    <T.GROSS(5)>"
GOTO UPDATE
ENDCASE
```

Note the D. prefix (display) that displays protected field values, and the T. prefix (turnaround) that displays field values to be updated. Display fields and turnaround fields are described in *Designing Screens With FIDEL* on page 227. Make all turnaround fields non-conditional; that is, end the field name with a right caret.

Once you have updated the values, you can transmit all the changes at one time by pressing the *Enter* key. These changes update the appropriate instances in the Scratch Pad Area. The request then branches to the modification phase (the UPDATE case), where your changes are entered into the data source. The CRTFORM may then prompt you for the next parent instance or may display the next set of multiple instances for you to change.

For example, a request that updates employee's monthly pay prompts you for an employee ID. This employee has eight pay dates recorded. The screen displays the first five pay dates. Make your adjustments and press *Enter*. The screen displays the last three pay dates. Make your adjustments and press *Enter*. The request then prompts you for the next employee ID.

You may add subscripts to fields only in CRTFORMs, not in REPEATs. REPEATs that follow the CRTFORMs process the fields in the order of the instances in the Scratch Pad Area, one at a time.

Procedure: How to Position the Cursor on Specific Field Values

You can design the request so that the cursor is automatically positioned on a particular field value on the FIDEL screen. To do this, set the CURSOR variable equal to the field name, as described in *Designing Screens With FIDEL* on page 227. If the fields are subscripted, set a field called CURSORINDEX equal to the value of the subscript. For example, this COMPUTE statement places the cursor on the field CURR_SAL(3):

```
COMPUTE
  CURSOR/A12 = 'CURR_SAL';
  CURSORINDEX = 3;
```

These cursor-positioning variables are useful when you perform validation tests on data entered on the FIDEL screen. After the CRTFORM, write a REPEAT statement for each field you are validating. Specify as many executions for the REPEAT as the highest subscript in the CRTFORM.

In the REPEAT statement:

- Set the CURSOR variable equal to the name of the field you are validating.
- Set the CURSORINDEX variable equal to the REPEATCOUNT variable. This sets the CURSORINDEX variable to the subscript of the field being validated.
- Validate the field.
- If a field value proves invalid, branch back to the CRTFORM using Case Logic. The CURSOR and CURSORINDEX variables will position the cursor at the invalid value.

Note: Remember to assign the CURSOR variable a format of A12 and the CURSORINDEX variable a format of I5.

This is a sample case validating the CURR_SAL field:

```
CASE DISPLAY
CRTFORM
"EMPLOYEE      SALARY            DEPARTMENT"
"--------      ------            --------- "
"<D.EMP_ID(1)  <T.CURR_SAL(1)>   <T.DEPARTMENT(1)>"
"<D.EMP_ID(2)  <T.CURR_SAL(2)>   <T.DEPARTMENT(2)>"
"<D.EMP_ID(3)  <T.CURR_SAL(3)>   <T.DEPARTMENT(3)>"
"<D.EMP_ID(4)  <T.CURR_SAL(4)>   <T.DEPARTMENT(4)>"
"<D.EMP_ID(5)  <T.CURR_SAL(5)>   <T.DEPARTMENT(5)>"

REPEAT 5 TIMES
  COMPUTE
    CURSOR/A12 = 'CURR_SAL';
    CURSORINDEX/15 = REPEATCOUNT;
  VALIDATE
    SALTEST = IF CURR_SAL GT 50000 THEN 0 ELSE 1;
    ON INVALID TYPE
       "THIS SALARY ENTERED WAS TOO HIGH"
       "PLEASE RE-ENTER"
    ON INVALID GOTO DISPLAY
ENDREPEAT
ENDCASE
```

The Modification Phase

After the user has entered changes on a FIDEL screen, the request uses the data to update instances in the Scratch Pad Area and to add new ones. To transfer the changes from the Area to the data source, prepare a REPEAT statement that modifies a data source instance on each pass.

This REPEAT updates the EMPLOYEE data source using data entered on the FIDEL screen shown in the previous section, *The Display Phase: Displaying Instances in One CRTFORM* on page 175. The REPEAT should execute as many times as there are instances in the Scratch Pad Area. This number was stored in the HOLDCOUNT variable. However, the HOLDCOUNT value can be changed by the CRTFORMs that display records in the Area. Therefore, you should store the HOLDCOUNT variable in a temporary field in the display phase before the CRTFORM. (This is shown in the example at the beginning of the section mentioned above.) This field can then set the number of times that the REPEAT statement executes.

At each pass, the REPEAT statement retrieves one instance from the Scratch Pad Area. It can then match on key fields in the instance to locate the corresponding instance in the data source (or determine that such an instance does not exist), then update the data source instance or add a new one.

In this example, the case UPDATE updates the data source instances, then branches back to the collection phase (COLLECT case). The collection phase reads the next five employee pay dates, which you can then change on the CRTFORM. This cycle continues until all the employee's pay dates have been read. You then enter the ID of the next employee. The number of instances in the Scratch Pad Area is contained in the temporary field BUFFNUMBER:

```
CASE UPDATE
REPEAT BUFFNUMBER
  MATCH PAY_DATE
    ON NOMATCH INCLUDE
    ON MATCH UPDATE GROSS
ENDREPEAT
GOTO COLLECT
ENDCASE

DATA VIA FI3270
END
```

Example: Using Multiple Record Processing (REPEAT Method)

The sample request on the next page updates the monthly pay of employees. The CRTFORM in the display phase displays the data for the five months in which the employee was paid. After you update the monthly pay of these five months, the display phase displays the next five months. This continues until it displays all the months recorded for that employee. The request then prompts for the next employee ID.

The request is as follows:

```
MODIFY FILE EMPLOYEE
CRTFORM LINE 2
"************************************"
"*MONTHLY PAY UPDATE*"
"************************************"
" "
"ENTER EMPLOYEE'S ID: <EMP_ID"

MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO COLLECT


CASE COLLECT
REPEAT 5 TIMES
  NEXT PAY_DATE
     ON NEXT HOLD PAY_DATE GROSS
     ON NONEXT GOTO DISPLAY
ENDREPEAT
GOTO DISPLAY
ENDCASE
```

```
CASE DISPLAY
IF HOLDCOUNT EQ 0 GOTO TOP;
COMPUTE
  BUFFNUMBER/I6=HOLDCOUNT;
CRTFORM LINE 9
" MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
" "
"PAY DATE               AMOUNT PAID"
"---------------        -----------"
"<D.PAY_DATE(1)         <T.GROSS(1)>"
"<D.PAY_DATE(2)         <T.GROSS(2)>"
"<D.PAY_DATE(3)         <T.GROSS(3)>"
"<D.PAY_DATE(4)         <T.GROSS(4)>"
"<D.PAY_DATE(5)         <T.GROSS(5)>"
GOTO UPDATE
ENDCASE
CASE UPDATE
REPEAT BUFFNUMBER
  MATCH PAY_DATE
    ON NOMATCH INCLUDE
    ON MATCH UPDATE GROSS
ENDREPEAT
GOTO COLLECT
ENDCASE
DATA VIA FI3270
END
```

Manual Methods

This section discusses manual methods of multiple record processing. These methods allow you to manipulate individual records in the Scratch Pad Area and to process instances from multiple segments at one time.

Manual methods depend on two temporary fields:

- The HOLDINDEX field. This field contains index values of records in the Scratch Pad Area. When you place a record in the Area using the HOLD statement, FOCUS assigns the record an index value equal to the value of the HOLDINDEX field. When you request a record from the Area using the GETHOLD statement, FOCUS retrieves the record having an index value equal to the value of the HOLDINDEX field.

  When you place a record into the area using the HOLD phrase, set HOLDCOUNT equal to HOLDINDEX, then increment HOLDINDEX by 1.

- The SCREENINDEX field. This field determines the group of records to appear on subscripted CRTFORMs.

There are manual methods for the collection, sorting, display, and modification phases of multiple record processing. There are no manual methods for the first phase, the selection phase (discussed in *Multiple Record Processing* on page 169). Note, however, that if you process multiple segments that have no common parent, you must select the parent instance of each segment chain.

Initialization

Before loading instances into the Scratch Pad Area, the request may need to perform the following tasks:

- Define the following variables with a format of I5:

  The HOLDCOUNT field. Set HOLDCOUNT equal to 0.

  The HOLDINDEX field. Set HOLDINDEX equal to 1.

  The SCREENINDEX field. Set SCREENINDEX equal to 0.

- Use the REPOSITION statement to insure that the current position in each segment, from which instances will be loaded into the Scratch Pad Area, is at the beginning of the segment.

The following is the beginning of a MODIFY request that uses manual methods:

```
MODIFY FILE EMPLOYEE
CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5 = 0;
  HOLDINDEX/I5 = 1;
  SCREENINDEX/I5 = 0;
REPOSITION SALARY
REPOSITION PAY_DATE
GOTO SALCOLLECT
ENDCASE
```

The Collection Phase: The HOLDINDEX Field

During the collection phase, the request retrieves multiple segment instances from the data source and stores each instance as a record in the Scratch Pad Area. FOCUS assigns each record an index value equal to the current value of the HOLDINDEX field, then increments HOLDINDEX by 1. For example, if HOLDINDEX is equal to 5, then the request stores one segment instance in the Area as Record 5, the next instance as Record 6, and so on.

To store instances from multiple segments, follow this procedure:

1. Assign each segment a range of index values (for example, assign one segment values 1 through 5, another 6 through 11, and so on).
2. Write the request so that a separate case loads instances from each segment. Before each case executes, have a COMPUTE statement set HOLDINDEX to the index value of the first record for that segment.

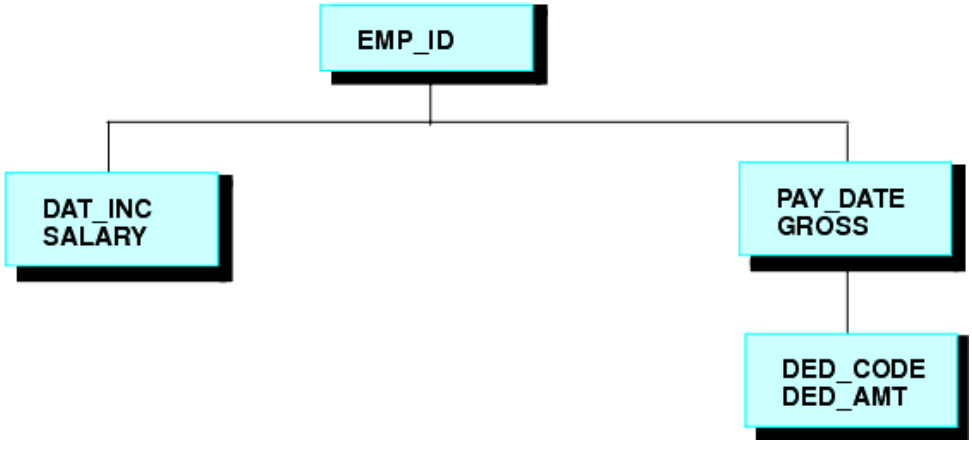
To assign index values to a segment, you must know the largest number of instances you will be storing from that segment. In many applications, you will be storing an entire segment chain at a time. You then must know the size of the largest segment chain.

Note: Be sure that you set HOLDINDEX to a value less than or equal to the current value of the HOLDCOUNT field. A HOLDINDEX value greater than HOLDCOUNT generates an error that terminates the request.

For example, suppose you write a request to update both employees' salary history (SALARY) and monthly pay (GROSS), information contained in two different segments in the EMPLOYEE data source (see the diagram that follows).

To determine the size of the largest chains in both segments, enter this procedure:

```
TABLE FILE EMPLOYEE
COUNT SALARY AND PAY_DATE BY EMP_ID
ON TABLE HOLD
END

TABLE FILE HOLD
SUM MAX.SALARY AND MAX.PAY_DATE
END
```



The output appears as follows:

```
PAGE    1
MAX       MAX
SALARY    PAY_DATE
------    --------
    2      10
```

The report shows that the largest salary history chain consists of two instances and the largest monthly pay chain consists of ten instances. Therefore, you assign values 1 and 2 to the salary history segment and values 3 through 12 to the monthly pay segment. Schematically, the Scratch Pad Area will look like this:

```
1. DAT_INC(1)   SALARY(1)         -              -
2. DAT_INC(2)   SALARY(2)         -              -
3.    -            -          PAY_DATE(3)      GROSS(3)
4.    -            -          PAY_DATE(4)      GROSS(4)
5.    -            -          PAY_DATE(5)      GROSS(5)
6.    -            -          PAY_DATE(6)      GROSS(6)
7.    -            -          PAY_DATE(7)      GROSS(7)
8.    -            -          PAY_DATE(8)      GROSS(8)
9.    -            -          PAY_DATE(9)      GROSS(9)
10.   -            -          PAY_DATE(10)     GROSS(10)
11.   -            -          PAY_DATE(11)     GROSS(11)
12.   -            -          PAY_DATE(12)     GROSS(12)
```

To fix the index values in the request, set HOLDINDEX to the first index value assigned to a segment before loading instances from that segment. In the example above, set HOLDINDEX to 1 before loading the salary history instances, and set HOLDINDEX to 3 before loading the monthly pay instances. This reserves the proper index values for each segment.

Prepare separate cases to load instances from each segment. During the modification phase, discussed on the next page, you may plan to retrieve all records from the same segment at one time. If so, store the index value of the last instance loaded into the Scratch Pad Area from that segment (this is the HOLDINDEX value after the last instance is loaded minus one) in a field. This field will help retrieve the records in the modification phase.

For example, you are loading monthly pay instances into the Scratch Pad Area. The last monthly pay instance loaded into the Area is assigned index value 8. You then store 8 in the field LASTPAY.

This example is a request fragment that updates employees' salary histories and monthly pay:

```
CASE SALCOLLECT
NEXT SALARY
  ON NEXT HOLD DAT_INC SALARY
  ON NEXT GOTO SALCOLLECT
  ON NONEXT COMPUTE
     LASTSAL/I5 = HOLDINDEX-1;
     HOLDINDEX = 3;
  ON NONEXT GOTO PAYCOLLECT
ENDCASE

CASE PAYCOLLECT
NEXT PAY_DATE
  ON NEXT HOLD PAY_DATE GROSS
  ON NEXT GOTO PAYCOLLECT
  ON NONEXT COMPUTE
     LASTPAY/I5 = HOLDINDEX-1;
  ON NONEXT GOTO DISPLAY
ENDCASE
```

The three cases are:

- **The TOP case.** This case selects an employee and sets the HOLDINDEX field to 1 to index the salary history instances.
- **The SALCOLLECT case.** This case loads the salary history instances into the Scratch Pad Area. After the instances are loaded, the case stores the index value of the last loaded salary history instance in the field LASTSAL. It then sets the HOLDINDEX field to 3 to index the monthly pay instances.

- **The PAYCOLLECT case.** This case loads the monthly pay instances into the Scratch Pad Area. After it loads the instances, it stores the index value of the last loaded monthly pay instance in the field LASTPAY. It then proceeds to the display phase.

The Display Phase: The SCREENINDEX Field

This section shows how to display a specific group of records in the Scratch Pad Area.

The REPEAT Method on page 170 described how to display records in the Scratch Pad Area on a CRTFORM. The CRTFORM statement specifies the field names with subscripts that refer to the records in the Area. For example:

```
CRTFORM
  "MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
  " "
  "PAY DATE          AMOUNT PAID"
  "--------          -----------"
  "<D.PAY_DATE(1)    <T.GROSS(1)>"
  "<D.PAY_DATE(2)    <T.GROSS(2)>"
  "<D.PAY_DATE(3)    <T.GROSS(3)>"
  "<D.PAY_DATE(4)    <T.GROSS(4)>"
  "<D.PAY_DATE(5)    <T.GROSS(5)>"
```

To display a subscripted field, FOCUS adds the field subscript to the value of a field called SCREENINDEX, then uses the sum as an index value to locate a record in the Scratch Pad Area. It then displays the field value in that record. For example, if the SCREENINDEX value for the above CRTFORM is 4, FOCUS will display the PAY_DATE and GROSS values from Area records 5 through 9.

You can use this feature to scroll back and forth through the Scratch Pad Area. To scroll forward, increase the value of SCREENINDEX; to scroll backward, decrease the value of SCREENINDEX.

If you update a field value on the CRTFORM, FOCUS updates the appropriate record in the Scratch Pad Area.

Note:

- If the request does not give SCREENINDEX a value, the default value is 0.
- If the sum of the SCREENINDEX value and a field subscript is less than 0 or more than the current value of the HOLDCOUNT field, then the CRTFORM displays that field as blank.
- If you use the CURSORINDEX field to place the cursor on a field value (as described in *The REPEAT Method* on page 170), the CURSORINDEX value refers to the field subscript, not the index value.

This sample case displays blocks of eight records stored in the Scratch Pad Area. The first record in each block is a monthly pay instance. The remaining seven records are deductions taken from the employee's paycheck. The case is:

```
CASE DISPLAY
IF HOLDCOUNT EQ 0 THEN GOTO TOP;
COMPUTE
  PFKEY/A4 = ' ';
  EMPID/A9 = EMP_ID;
  DED_AMT/D12.2M = DED_AMT;
CRTFORM LINE 1
  "DEDUCTION RECORD SCREEN"
  " "
"  EMPLOYEE: <D.EMPID      PAY DATE: <D.PAY_DATE(1)"
" "
"1. <D.DED_CODE(2)    <T.DED_AMT(2)>"
"2. <D.DED_CODE(3)    <T.DED_AMT(3)>"
"3. <D.DED_CODE(4)    <T.DED_AMT(4)>"
"4. <D.DED_CODE(5)    <T.DED_AMT(5)>"
"5. <D.DED_CODE(6)    <T.DED_AMT(6)>"
"6. <D.DED_CODE(7)    <T.DED_AMT(7)>"
"7. <D.DED_CODE(8)    <T.DED_AMT(8)>"
" "
"PRESS PF4 TO DISPLAY THE NEXT EMPLOYEE"
"PRESS PF5 TO DISPLAY THE LAST PAY DATE"
"PRESS PF6 TO DISPLAY THE NEXT PAY DATE"
COMPUTE
  SCREENINDEX/I5 = IF PFKEY IS 'PF04' THEN 0
    ELSE IF PFKEY IS 'PF05' THEN SCREENINDEX - 8
    ELSE IF PFKEY IS 'PF06' THEN SCREENINDEX + 8
    ELSE SCREENINDEX;
IF PFKEY IS 'PF04' THEN GOTO TOP ELSE GOTO DISPLAY;
```

Pressing one of the PF keys gives the variable PFKEY a value that the request tests to adjust SCREENINDEX. By adding eight to SCREENINDEX, the request displays the next block of records. By subtracting eight from SCREENINDEX, the request displays the previous block of records.

The Modification Phase: The GETHOLD Statement

During the modification phase, the request retrieves records from the Scratch Pad Area and uses them to modify the data source. It retrieves records using the GETHOLD statement. The syntax is

```
GETHOLD
```

without any parameters. The GETHOLD statement retrieves the record whose index value is the value of the HOLDINDEX field. The HOLDINDEX field is then incremented by 1. For example, if the current value of HOLDINDEX is 5, the GETHOLD statement retrieves Record 5 from the Scratch Pad Area. HOLDINDEX is then increased to 6.

After the record is retrieved, all fields in the record become available for processing: matching, adding new segment instances, updating, deleting, and computations. Note that you may need to activate these fields before processing. For example, these statements update an employee's monthly pay using Record 5 in the Scratch Pad Area. Record 5 contains two fields: PAY_DATE and GROSS:

```
COMPUTE HOLDINDEX = 5;
GETHOLD
ACTIVATE PAY_DATE GROSS
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON MATCH UPDATE GROSS
```

You may use the GETHOLD statement to process all the records in the Scratch Pad Area. If the records contain data loaded from different segments, use separate cases to process records from each segment. First, set the HOLDINDEX field to the index value of the first record from the segment. As the request retrieves each record, HOLDINDEX increases by 1. When HOLDINDEX is greater than the index value of the last record from the segment (which you stored earlier in a field), you can branch to another case.

For example, this request fragment updates employees' salary history and monthly pay. The Scratch Pad Area consists of the following records:

- The first two records contain the fields DAT_INC and SALARY to update the salary history.
- The next ten records contain the fields PAY_DATE and GROSS to update monthly pay.

The fragment is:

```
CASE SALSET
COMPUTE HOLDINDEX = 1;
GOTO SALUPDATE
ENDCASE

CASE SALUPDATE
GETHOLD
MATCH DAT_INC
  ON MATCH UPDATE SALARY
  ON MATCH    IF HOLDINDEX GT LASTSAL GOTO PAYSET
     ELSE GOTO SALUPDATE;
  ON NOMATCH REJECT
ENDCASE

CASE PAYSET
COMPUTE HOLDINDEX = 3;
GOTO PAYUPDATE
ENDCASE

CASE PAYUPDATE
GETHOLD
MATCH PAY_DATE
  ON MATCH UPDATE GROSS
  ON MATCH    IF HOLDINDEX GT LASTPAY GOTO TOP
     ELSE GOTO PAYUPDATE;
  ON NOMATCH REJECT
ENDCASE

DATA VIA FIDEL
END
```

The cases are as follows:

- The SALSET case sets HOLDINDEX to 1, the index value of the first salary history record.
- The SALUPDATE case updates the salary history using the records in the Scratch Pad Area. Each time the case retrieves a record, HOLDINDEX is incremented by 1. When HOLDINDEX is greater than the index value of the last salary history record (the value of field LASTSAL), the case branches to the PAYSET case.
- The PAYSET case sets HOLDINDEX to 3, the index value of the first monthly pay record in the Scratch Pad Area.
- The PAYUPDATE case updates monthly pay using the records in the Scratch Pad Area. When HOLDINDEX is greater than the index value of the last monthly pay record in the Area (the value of field LASTPAY), the case branches back to the top.

You can also use the GETHOLD statement to retrieve and process a single record from the Scratch Pad Area. This request fragment allows the user to delete a single monthly pay instance:

```
CASE DISPLAY
CRTFORM
COMPUTE LN/I1 = 0;
  "MONTHLY PAY FOR <D.FIRST_NAME <D.LAST_NAME"
  " "
  "PAY DATE           AMOUNT PAID"
  "--------           -----------"
  "1. <D.PAY_DATE(1)  <T.GROSS(1)>"
  "2. <D.PAY_DATE(2)  <T.GROSS(2)>"
  "3. <D.PAY_DATE(3)  <T.GROSS(3)>"
  "4. <D.PAY_DATE(4)  <T.GROSS(4)>"
  "5. <D.PAY_DATE(5)  <T.GROSS(5)>"
  " "
  "TO DELETE AN INSTANCE, ENTER LINE NUMBER HERE: <LN"
IF (LN LT 1) OR (LN GT 5) GOTO DISPLAY ELSE GOTO DELETE;
ENDCASE

CASE DELETE
COMPUTE
  HOLDINDEX = LN;
GETHOLD
MATCH PAY_DATE
  ON NOMATCH REJECT
  ON NOMATCH GOTO TOP
  ON MATCH DELETE
  ON MATCH GOTO TOP
ENDCASE
```

Note: Be sure that you set HOLDINDEX to a value less than or equal to the current value of the HOLDCOUNT field. A HOLDINDEX value greater than HOLDCOUNT generates an error that terminates the request.

Reference:

Manual Methods: Two Examples

This section shows two examples that illustrate manual methods in multiple record processing:

- The first example updates employees' salary history and monthly pay. This is data contained in segments on two different paths.
- The second example deletes records of employee deductions. This is data contained in segments on one path (a parent and its child).

A diagram showing the place of salary history (SALARY), monthly pay (GROSS), and pay deductions (DED_AMT) in the EMPLOYEE data source structure appears at the beginning of *The Collection Phase: The HOLDINDEX Field* on page 182 in this section.

Example: First Example: Processing Segments on Two Different Paths

This request is an example of a procedure that processes segments lying on different paths. The example updates employees' salary history and monthly pay. The salary history segment and monthly pay segment are both children of the employee segment, and they are on two separate paths.

This request also demonstrates the use of the GETHOLD statement to retrieve segment chains from the Scratch Pad Area. Explanatory comments are embedded in the request.

```
MODIFY FILE EMPLOYEE
-* First, select the parent employee instance.

CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
-* Flush the Scratch Pad Area, then initialize fields
-* and segment chains.

REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5 = 0;
  HOLDINDEX/I5 = 1;
REPOSITION SALARY
REPOSITION PAY_DATE
GOTO SALCOLLECT
ENDCASE
```

```
CASE SALCOLLECT
-* Place the employees' salary history in the Scratch
-* Pad Area. Afterwards, store the index value of the
-* last loaded instance in the field LASTSAL. Then
-* set HOLDINDEX to 3, which is the index of the
-* first monthly pay instance.

NEXT SALARY
  ON NEXT HOLD DAT_INC SALARY
  ON NEXT GOTO SALCOLLECT
  ON NONEXT COMPUTE
     LASTSAL/I5 = HOLDINDEX-1;
  HOLDINDEX = 3;
  ON NONEXT GOTO PAYCOLLECT
ENDCASE

CASE PAYCOLLECT
-* Place the monthly pay instances in the Scratch Pad
-* Area. Afterwards, store the index value of the last
-* loaded instance in the field LASTPAY.

NEXT PAY_DATE
  ON NEXT HOLD PAY_DATE GROSS
  ON NEXT GOTO PAYCOLLECT
  ON NONEXT COMPUTE
     LASTPAY/I5 = HOLDINDEX-1;
  ON NONEXT GOTO DISPLAY
ENDCASE

CASE DISPLAY
-* If nothing was collected, go back to TOP.
-* Otherwise, display the two segment chains
-* side by side. Then reset HOLDINDEX to 1
-* to prepare for updating.

IF HOLDCOUNT EQ 0 GOTO TOP;
CRTFORM LINE 3
  "SALARY HISTORY AND MONTHLY PAY RECORD"
  " "
  "SALARY HISTORY               <40 MONTHLY PAY"
  --------------                <40 -----------"
  " "
  " <D.DAT_INC(1) <T.SAL(1>         <40 <D.PD(3)  <T.GROSS(3)>"
  " <D.DAT_INC(2) <T.SAL(2>         <40 <D.PD(4)  <T.GROSS(4)>"
  "                                 <40 <D.PD(5)  <T.GROSS(5)>"
  "                                 <40 <D.PD(6)  <T.GROSS(6)>"
  "                                 <40 <D.PD(7)  <T.GROSS(7)>"
  "                                 <40 <D.PD(8)  <T.GROSS(8)>"
  "                                 <40 <D.PD(9)  <T.GROSS(9)>"
  "                                 <40 <D.PD(10) <T.GROSS(10)>"
  "                                 <40 <D.PD(11) <T.GROSS(11)>"
  "                                 <40 <D.PD(12) <T.GROSS(12)>"
COMPUTE HOLDINDEX=1;
GOTO SALUPDATE
ENDCASE

CASE SALUPDATE
-* Update the salary history instances.
-* LASTSAL contains the index value of the
-* last salary history record.

GETHOLD
MATCH DAT_INC
  ON MATCH UPDATE SALARY
```

Example: Second Example: Modifying Segments on the Same Path

This is a sample request that processes segments lying on the same path. The request deletes employee pay deductions. To do so, it displays a pay date on the top of the screen; below, it shows the deductions taken from the employee's pay check that date. The user can scroll back and forth between pay dates and may choose particular deductions to delete. The pay date is a field in the monthly pay segment; the deductions are fields in the child deduction segment, as shown in the diagram in *The Collection Phase: The HOLDINDEX Field* on page 182.

The request also demonstrates the use of the SCREENINDEX field to display different groups of records on subscripted CRTFORMs, and the use of the GETHOLD statement to retrieve specific records. Explanatory comments are embedded in the request.

```
MODIFY FILE EMPLOYEE

-* First, select the parent employee instance.

CRTFORM
  "ENTER EMPLOYEE ID: <EMP_ID"
MATCH EMP_ID
  ON NOMATCH REJECT
  ON MATCH GOTO INITIAL

CASE INITIAL
-* Flush the Scratch Pad Area, then initialize fields
-* and segment chains.

REPEAT 1
  HOLD EMP_ID
ENDREPEAT
COMPUTE
  HOLDCOUNT/I5    = 0;
  HOLDINDEX/I5    = 1;
  SCREENINDEX/I5  = 0;
BLOCKCOUNT/I5     = 0;
REPOSITION PAY_DATE
GOTO PAYCOLLECT
ENDCASE
```

```
CASE PAYCOLLECT
-* The next two cases create blocks of eight
-* instances within the Scratch Pad Area. Each block
-* consists of a monthly pay instance followed
-* by seven descendant instances in the
-* deduction segment. The field BLOCKCOUNT counts
-* the number of blocks in the Scratch Pad Area so far.
-* The field BLOCKNUM contains the total number of
-* blocks in the Area after all instances have
-* been loaded.

NEXT PAY_DATE
  ON NEXT COMPUTE
     HOLDINDEX = 8 * BLOCKCOUNT + 1;
     BLOCKCOUNT = BLOCKCOUNT + 1;
  ON NEXT ACTIVATE PAY_DATE
  ON NEXT HOLD PAY_DATE
  ON NEXT GOTO DEDCOLLECT
  ON NONEXT COMPUTE
     BLOCKNUM/I5 = BLOCKCOUNT;
  ON NONEXT GOTO DISPLAY
ENDCASE

CASE DEDCOLLECT
NEXT DED_CODE
  ON NEXT ACTIVATE DED_CODE DED_AMT
  ON NEXT HOLD DED_CODE DED_AMT
  ON NEXT GOTO DEDCOLLECT
  ON NONEXT GOTO PAYCOLLECT
ENDCASE

CASE DISPLAY
-* If nothing was collected, go back to TOP.
-* Otherwise, initialize the PFKEY and LINENO
-* fields. The EMPID field is for display
-* purposes. Then, display the current block.
-*
-* At the bottom of the screen is a menu to offer
-* users the choice of processing the records
-* of another employee, displaying the previous
-* block or displaying the next block. the field
-* PFKEY reads the PF key that the user presses
-* (see Chapter 16). The field LINENO contains the
-* line number of the deduction instance that the
-* user wants to delete.

IF HOLDCOUNT EQ 0 THEN GOTO TOP;
COMPUTE
  PFKEY/A4     = ' ';
  LINENO/I1    = 0;
  EMPID/A9     = EMP_ID;
CRTFORM LINE 1
  "DEDUCTION RECORD DELETION SCREEN"
  " "
  "EMPLOYEE: <D.EMPID     PAY DATE: <D.PAY_DATE(1)"
  " "
  "1. <D.DED_CODE(2) <D.DED_AMT(2)"
  "2. <D.DED_CODE(3) <D.DED_AMT(3)"
  "3. <D.DED_CODE(4) <D.DED_AMT(4)"
  "4. <D.DED_CODE(5) <D.DED_AMT(5)"
  "5. <D.DED_CODE(6) <D.DED_AMT(6)"
  "6. <D.DED_CODE(7) <D.DED_AMT(7)"
  "7. <D.DED_CODE(8) <D.DED_AMT(8)"
  " "
```

Procedure: How to Sort the Scratch Pad Area With SORTHOLD

You can sort the contents of the Scratch Pad Area using any field or combination of fields in the Scratch Pad Area; you can then display them in any convenient order. The command uses syntax similar to the sorting specifications in the TABLE command.

The MODIFY subcommand that sorts the Scratch Pad Area is

```
SORTHOLD BY [HIGHEST] field1 [BY [HIGHEST] field2...]
```

where *field1* is the primary sort field, and *field2* to *field8* are optional secondary sort fields.

Note:

- The SORTHOLD statement cannot span more than one line. The default sort order is from low-to-high, but a high-to-low sort can be specified with the keyword HIGHEST. You can sort the Scratch Pad Area by up to eight fields.
- If you sort the Scratch Pad Area before display, always sort by the data source key fields before entering a MATCH... UPDATE loop, to be sure the transactions are in sequence with the data source. Otherwise you increase execution time substantially. This procedure

  ```
  SORTHOLD BY ITEM
  ```

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

Consider the following Master File:

```
FILENAME=PRODUCT, SUFFIX=FOC
  SEGNAME=SEGONE, SEGTYPE=S1
   FIELD=ORDERNO,     ALIAS=ONO,     FORMAT=I4,  $
  SEGNAME=SEGTWO, SEGTYPE=S1, PARENT=SEGONE
   FIELD=ITEM,        ALIAS=ITEMNO, FORMAT=A3,  $
   FIELD=PRODUCT,     ALIAS=PRD,     FORMAT=A12, $
   FIELD=QTY,    ALIAS=QUANTITY,     FORMAT=I4S, $
```

The following procedure will display all of the ITEM instances for a specified ORDERNO, in order of the PRODUCT name and highest QTY sequence. The command

```
SORTHOLD BY PRODUCT BY QTY
```

performs the sort.

```
MODIFY FILE PRODUCT
CRTFORM LINE 1
  "ENTER ORDER NUMBER <ORDERNO"
MATCH ORDERNO
  ON NOMATCH GOTO TOP
  ON MATCH REPEAT 12
     NEXT ITEM
       ON NEXT HOLD ITEM PRODUCT QTY
       ON NONEXT GOTO SCREEN
  ENDREPEAT
GOTO SCREEN
CASE SCREEN
  IF HOLDCOUNT EQ 0 GOTO TOP;

  SORTHOLD BY PRODUCT BY HIGHEST QTY

  CRTFORM LINE 1
    "ORDER NUMBER IS <D.ORDERNO          "
  "                                      "
  " ITEM       PRODUCT          QUANTITY "
  " ----       -------          -------- "
  "<D.ITEM(1)  <D.PRODUCT(1)  <T.QTY(1)> "
  "<D.ITEM(2)  <D.PRODUCT(2)  <T.QTY(2)> "
     .
     .
     .
  "<D.ITEM(12) <D.PRODUCT(12) <T.QTY(12)>"

  SORTHOLD BY ITEM
  REPEAT HOLDCOUNT
    MATCH ITEM
      ON MATCH UPDATE
QTY
      ON NOMATCH GOTO
ENDREPEAT
  ENDREPEAT
  GOTO TOP
ENDCASE
DATA VIA FIDEL
END
```

Advanced Facilities

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

- The COMBINE command, for modifying multiple FOCUS data sources in one MODIFY request.
- The COMPILE command, for translating MODIFY requests into compiled code ready for execution.
- The ACTIVATE and DEACTIVATE statements, for activating and deactivating fields.

- The Checkpoint and Absolute File Integrity facilities, for protecting FOCUS data sources from system failures.
- The ECHO facility, for displaying the logical structure of MODIFY requests.
- Dialogue Manager system variables, which record execution statistics every time a MODIFY request is run.
- FOCUS query commands, which display statistical information on MODIFY request executions and FOCUS data sources.
- COMMIT and ROLLBACK subcommands, for controlling changes made to FOCUS data sources, and for protecting FOCUS data sources from system failures.

All these facilities are described in the sections that follow.

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

Modifying Multiple Data Sources in One Request: The COMBINE Command

The COMBINE command allows you to modify two or more FOCUS, relational, or Adabas data sources in the same MODIFY request. The command combines the logical structures of the FOCUS data sources into one structure while leaving the physical structures of the data sources untouched. This combined structure lasts for the duration of the FOCUS session, until you enter another COMBINE command, or it is cleared with the AS CLEAR option. Only one combined structure can exist at a time.

Note the following:

- The combined structure can contain up to 63 segments from up to 63 data sources with one additional reserved for BINS.
- You can COMBINE data sources that come from different applications and have different DBA passwords. The only requirement is a valid password for each data source. For more information, refer to the *Describing Data* manual.
- Only the MODIFY and CHECK commands can process combined structures.
- If you are using Simultaneous Usage mode, all the data sources in the combined structure must either be all on the same FOCUS Database Server or all in local mode.
- All MODIFY code compiled in releases prior to 5.2.0 must be re-compiled.
- The differences between JOIN and COMBINE commands are discussed in *Differences Between COMBINE and JOIN Commands* on page 203.

Syntax: How to Combine Data Sources

Enter the COMBINE command at the FOCUS command level (at the FOCUS prompt).

```
COMBINE FILES file1 [PREFIX pref1|TAG tag1] [AND]
   .
   .
   .
            filen [PREFIX prefn|TAG tagn] AS asname
```

where:

file1... filen

Are the Master File names for the data sources you want to modify. You can specify up to 63 data sources (you will be limited to fewer data sources if any of these data sources have more than one segment).

pref1... prefn

Are prefix strings for each data source; up to four characters. They provide uniqueness for field names. You cannot mix TAG and PREFIX in a COMBINE structure. See *Referring to Fields in Combined Structures: The PREFIX Parameter* on page 200 later in this section.

tag1... tagn

Are aliases for the Master File names; up to eight characters. FOCUS uses the tag name as the qualifier for fields that refer to that data source in the combined structure. You cannot mix TAG and PREFIX in a COMBINE, and you can only use TAG if FIELDNAME is set to NEW or NOTRUNC. See *Referring to Fields in Combined Structures: The TAG Parameter* on page 199 later in this section.

AND

Is an optional word to enhance readability.

asname

Is the required name of the combined structure to use in MODIFY procedures and CHECK FILE commands. For example, if you name the combined structure EDJOB, begin the request with:

```
MODIFY FILE EDJOB
```

AS CLEAR

Is the command that clears the combined structure which is currently in effect.

Note: The AS CLEAR option must be issued with no file name:

```
COMBINE FILE AS CLEAR
```

Once you enter the COMBINE command, you can modify the combined structure.

Note:

- TAG and PREFIX may not be used together in a COMBINE.
- You can type the command on one line or on as many lines as you need.

Example: COMBINE Command

For example, to combine data sources EDUCFILE and JOBFILE, enter:

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

After entering this command, you can run the following request. Notice that the statements pertaining to each data source are placed in different cases (Case Logic is discussed in *Case Logic* on page 145). This clarifies the request logic, and makes it easier to understand and clarify the request. The first case modifies the EDUCFILE data source, and the second case modifies the JOBFILE data source.

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


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


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

Syntax: How to Support Long and Qualified Field Names

If you are using tag names, you must also set the command SET FIELDNAME to NEW or NOTRUNC. The SET FIELDNAME command enables you to activate long (up to 66 characters) and qualified field names. The syntax for this SET command is

```
SET FIELDNAME = type
```

where:

type
Is one of the following:

OLD specifies that 66-character and qualified field names are not supported; the maximum length is 12 characters.

NEW specifies that 66-character and qualified field names are supported; the maximum length is 66 characters. NEW is the default value.

NOTRUNC prevents unique truncations of field names and supports the 66-character maximum.

When the value of FIELDNAME is changed within a FOCUS session, COMBINE commands are affected as follows:

- When you change from a value of OLD to a value of NEW, all COMBINE commands are cleared.
- When you change from a value of OLD to NOTRUNC, all COMBINE commands are cleared.
- When you change from a value of NEW to OLD, all COMBINE commands are cleared.
- When you change from a value of NOTRUNC to OLD, all COMBINE commands are cleared.

Other changes to the FIELDNAME value do not affect COMBINE commands.

Note: For more information on the SET FIELDNAME command, refer to the *Developing Applications* manual.

Reference: Referring to Fields in Combined Structures: The TAG Parameter

For a MODIFY request to refer to transaction fields in a combined structure by their transaction field names, the field names must be unique; that is, the transaction field names in one data source cannot appear in other data sources. Refer to any transaction field names that are not unique by their aliases, or use the TAG parameter in the COMBINE command to assign a tag name to the data sources that share the transaction field names.

When a data source has a tag, refer to its transaction field names by affixing the tag name to the beginning of each field name.

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

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

When you create a request that modifies this structure, type the EDUCFILE field names with the AAA prefix in front:

```
COMBINE FILES EDUCFILE TAG AAA AND JOBFILE AS EDJOB
MODIFY FILE EDJOB
PROMPT AAA.COURSE_CODE AAA.COURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE
CASE EDUCFILE
MATCH AAA.COURSE_CODE
ON MATCH REJECT
ON NOMATCH INCLUDE
GOTO JOBFILE
ENDCASE
CASE JOBFILE
MATCH JOBCODE
ON MATCH REJECT
ON NOMATCH INCLUDE
ENDCASE
DATA
```

In this request, the tag AAA has been attached to the two transaction field names in the EDUCFILE data source: COURSE_CODE and COURSE_NAME, making the new field names AAA.COURSE_CODE and AAA.COURSE_NAME. Use these tagged field names only in MODIFY requests that modify the combined structure.

Reference: Referring to Fields in Combined Structures: The PREFIX Parameter

For a MODIFY request to refer to fields in a combined structure by their field names, the field names must be unique so that there is no ambiguity in the request. That is, the field names in one data source cannot appear in other data sources. If there are field names that are not unique, refer to the fields by their aliases or use the PREFIX parameter in the COMBINE command to assign a prefix of up to four characters to the data sources sharing the field names.

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

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

When you enter a request modifying the structure, type the EDUCFILE field names with the ED prefix in front:

```
COMBINE FILES EDUCFILE PREFIX ED JOBFILE AS EDJOB
MODIFY FILE EDJOB
PROMPT EDCOURSE_CODE EDCOURSE_NAME JOBCODE JOB_DESC
GOTO EDUCFILE
```

```
CASE EDUCFILE
MATCH EDCOURSE_COD
  ON MATCH REJECT
  ON NOMATCH INCLUDE
GOTO JOBFILE
ENDCASE

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

In this request, the prefix ED has been attached to the two field names in the EDUCFILE data source: COURSE_CODE and COURSE_NAME. The new field names are EDCOURSE_CODE and EDCOURSE_NAME.

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

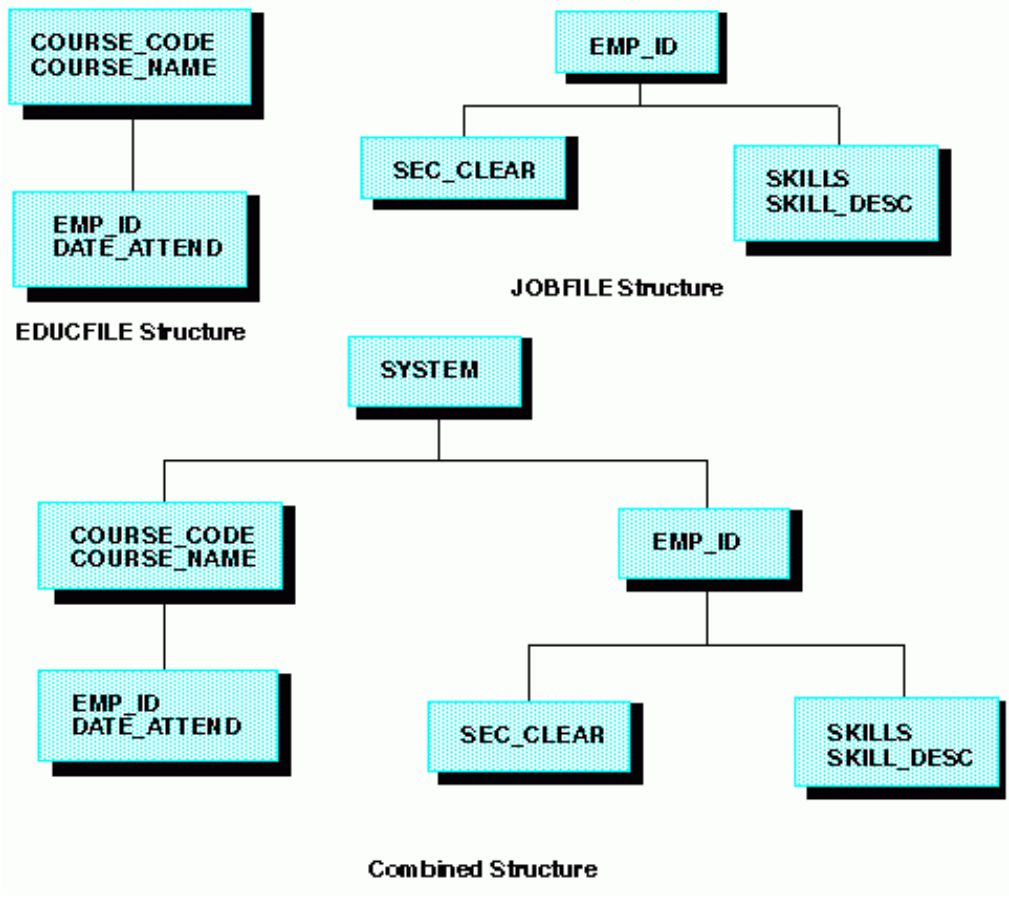
Note: A MODIFY COMBINE with prefixes cannot be loaded through the LOAD facility. However, the unloaded versions will run.

For more information on compiling MODIFY requests see *Compiling MODIFY Requests: The COMPILE Command*. For more information on loading data sources, see the *Developing Applications* manual.

Procedure: How to How Data Source Structures Are Combined

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

The following figure shows how two data sources, EDUCFILE and 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 data source does not change.



Field names are considered duplicates when two or more fields are referenced with the same field name or alias. Duplication can occur if a COMBINE is done without a prefix or a tag. Duplicate fields are not allowed in the same segment. The second occurrence is never accessed by FOCUS and the following warning message is generated when CHECK and CREATE FILE are issued:

```
(FOC1829) WARNING. FIELDNAME IS NOT UNIQUE WITHIN A SEGMENT: fieldname
```

Differences Between COMBINE and JOIN Commands

The COMBINE command differs from the JOIN command in the following ways:

- The JOIN command is effective for TABLE, TABLEF, MATCH, GRAPH, and CHECK commands, but is not effective for MODIFY requests (except for the LOOKUP function). The COMBINE command is effective only for MODIFY requests and CHECK commands.
- The JOIN command joins a variety of FOCUS and non-FOCUS data sources. The COMBINE command combines FOCUS data sources only.
- The JOIN command can only join data sources with common fields. The COMBINE command can combine all FOCUS data sources.
- The JOIN command joins data source structures together at segments with a common field. This can invert some of the segment relationships in the cross-referenced data source (see alternate file view in the *Describing Data* and *Creating Reports* manuals). The COMBINE command combines the data source structures under a dummy root segment. Segment relationships remain intact.

Syntax: How to Use the ? COMBINE Query

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

```
? COMBINE
```

FOCUS responds

```
FILE=name TAG  PREFIX
file-1 tag-1 prefix-1
file-2 tag-2 prefix-2
file-3 tag-3 prefix-3
. . .
. . .
file-n tag-n prefix-n
```

where:

name

Is the name of the combined structure.

file-1 ... file-n

Are the names of the data sources that make up the combined structure.

tag-1 ... tag-n

Are the tags attached to the field names in the data source. These tags correspond to the aliases given to the data source(s) in the combined structure.

prefix-1 ... prefix-n

Are the prefixes attached to the field names in the data source.

The ? COMBINE query shows up to 63 entries.

For example, when data source EDUCFILE is combined with data source JOBFILE, enter the command

```
? COMBINE
```

to display the following information:

```
COMBINE EDUCFILE AND JOBFILE AS EDJOB
>
? COMBINE
 FILE=EDJOB              TAG             PREFIX

      EDUCFILE
      JOBFILE
>
```

Note: TAG and PREFIX may not be mixed in a COMBINE.

Reference: **Error Messages for COMBINE**

```
(FOC???) MAXIMUM NUMBER OF 'COMBINES' EXCEEDED. CLEAR SOME AND RE-ENTER:
```

The number of separate COMBINE commands exceeds the current limit of 63.

Active and Inactive Fields

This section discusses active and inactive fields. When you run a request, FOCUS keeps track of which transaction fields are active or inactive during execution:

- Active fields have incoming data for them. You may use active fields to add, update, and delete segment instances.
- Inactive fields do not have incoming data for them. You can use inactive fields in calculations only.

When a MATCH statement matches on an inactive field, the request returns to the beginning (the TOP case in case requests) to avoid modifying segments for which data is not present.

If a MATCH or NEXT statement executes an INCLUDE action, all segment instances having active fields are added to the data source.

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

This section covers the following:

- When fields are active and inactive.
- Activating fields with the ACTIVATE statement.
- Deactivating fields with the DEACTIVATE statement.

Reference:

When Fields Are Active and Inactive

A data field becomes active when:

- It is described in the Master File and it is read in by a FIXFORM, FREEFORM, PROMPT, or CRTFORM statement. Note that if the field is declared a conditional field, the following rules apply:
  - In a FIXFORM statement, a conditional field is active when it has a value present in a record.
  - In a CRTFORM, a conditional entry field is active when you enter data for it. A conditional turnaround field is active when you change its value (see *Designing Screens With FIDEL* on page 227).
- The field is assigned a value by a COMPUTE or VALIDATE statement.
- The field is activated by the ACTIVATE statement.

A data field becomes inactive when:

- Execution branches to the top of the request, whether this is done implicitly or by a GOTO statement.
- It modifies a segment instance because of an INCLUDE, UPDATE, or DELETE action.
- It has been made available to the request through the LOOKUP function.
- It is deactivated by the DEACTIVATE statement.

Procedure: How to Activate Fields With the ACTIVATE Statement

To activate an inactive field, use the ACTIVATE statement. the ACTIVATE statement performs two tasks:

- It declares a transaction field to be present (considered part of the current transaction). The field can then be used for matching, including, and updating.
- It equates the value of the transaction field to the corresponding data source field. This occurs when both of the following conditions are true:
  - The ACTIVATE statement either appears within or it follows a MATCH or NEXT statement that modifies the segment containing the corresponding data source field.
  - The ACTIVATE statement converts the field from being inactive to active. Included are fields for which the request has not read any data or assigned a value with a compute statement. Fields already active are excluded.

If one of these conditions is not true, the activate statement does not change the value of the field. If the field has no data, FOCUS sets the value of the field to blank if alphanumeric, zero if numeric, and the missing data symbol if the field is described by the MISSING=ON attribute in the Master File (discussed in the *Describing Data* manual).

The syntax of the ACTIVATE statement is

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

where:

`RETAIN`

Is an option that activates the field but leaves its value unchanged, even if the ACTIVATE statement converts the field from being inactive to active.

`MOVE`

Is an option that activates the field and equates its value to the corresponding data source field, even if the field was already active before the ACTIVATE statement.

`field1 ...`

Are the names of the fields you want to activate. To activate all the fields in one segment, specify any segment field with the prefix SEG. affixed in front of the field name. For example:

```
ACTIVATE SEG.SKILLS
```

This sample request illustrates how ACTIVATE statements affect the fields they specify. The numbers on the margin refer to the notes below. The request is:

```
    MODIFY FILE EMPLOYEE

1.  FREEFORM EMP_ID CURR_SAL ED_HRS

2.  ACTIVATE DEPARTMENT
    MATCH EMP_ID
      ON MATCH REJECT
3.    ON NOMATCH INCLUDE
4.  GOTO NEXT_EMP1

    CASE NEXT_EMP1
5.  NEXT EMP_ID
      ON NONEXT GOTO EXIT
6.    ON NEXT ACTIVATE RETAIN CURR_SAL DEPARTMENT
7.    ON NEXT UPDATE DEPARTMENT ED_HRS
8.    ON NEXT GOTO NEXT_EMP2
    ENDCASE

    CASE NEXT_EMP2
9.  NEXT EMP_ID
      ON NONEXT GOTO EXIT
10.   ON NEXT ACTIVATE CURR_SAL DEPARTMENT ED_HRS
11.   ON NEXT ACTIVATE MOVE CURR_SAL
12.   ON NEXT GOTO NEXT_EMP3
    ENDCASE

    CASE NEXT_EMP3
13. NEXT EMP_ID
      ON NONEXT GOTO EXIT
14.   ON NEXT UPDATE CURR_SAL DEPARTMENT ED_HRS
    ENDCASE

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

When you run the request, the following happens:

1. The request reads the record:

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

2. The statement

   ```
   ACTIVATE DEPARTMENT
   ```

   activates the DEPARTMENT field. Since the request did not read any data for this field and the statement precedes the MATCH and NEXT statements, FOCUS equates the field value to blank.

   The transaction record is as follows:

```
Transaction Record:

EMP_ID: 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 data source. It therefore includes the record in the data source as a new segment instance. All fields included in the instance, EMP_ID, CURR_SAL, DEPARTMENT and ED_HRS, become inactive.
4. The request branches to the NEXT_EMP1 case.
5. The request moves the current position in the data source to the next segment instance after EMP_ID 444. This instance contains the following fields:

```
Database Segment Instance:

EMP_ID: 326179357
CURR_SAL: 21780.00
ED_HRS: 75.00
DEPARTMENT: MIS
```

6. The statement

```
ACTIVATE RETAIN CURR_SAL DEPARTMENT
```

activates the CURR_SAL and DEPARTMENT fields. The RETAIN keyword prevents their values from changing. The transaction record is now:

```
Transaction Record:

EMP_ID: 326179357 (inactive)
CURR_SAL: 50000 (active)
DEPARTMENT: blank (active)
ED_HRS: 40 (inactive)
```

7. The statement

```
UPDATE DEPARTMENT ED_HRS
```

changes the DEPARTMENT field value in the segment instance to blank and deactivates the DEPARTMENT field on the transaction record. Since the ED_HRS transaction field is inactive, it does not change the data source ED_HRS value. The segment instance is now:

```
Database Segment Instance:

EMP_ID: 326179357
CURR_SAL: 21780.00
DEPARTMENT: blank
ED_HRS: 75.00
```

The request did not use the CURR_SAL transaction field to update the instance, so the CURR_SAL field remains active. The transaction record is as follows:

```
Transaction Record:

EMP_ID: 326179357 (inactive)
CURR_SAL: 50000 (active)
DEPARTMENT: BLANK (inactive)
ED_HRS: 40 (inactive)
```

8. The request branches to the NEXT_EMP2 case.
9. The request moves the current position to the next current instance after EMP_ID 326179357. This instance contains the following fields:

```
Database Segment Instance:

EMP_ID: 451123478
CURR_SAL: 16100.00
DEPARTMENT: PRODUCTION
ED_HRS: 50.00
```

10. The statement

```
ACTIVATE CURR_SAL DEPARTMENT ED_HRS
```

declares the CURR_SAL, DEPARTMENT, and ED_HRS transaction fields to be active. Since CURR_SAL was already active, its value does not change. DEPARTMENT and ED_HRS are converted into active fields, and their values change to that of the DEPARTMENT and ED_HRS fields in the segment instance. The transaction record is now:

```
Transaction Record:

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

11. The statement

```
ACTIVATE MOVE CURR_SAL
```

declares the CURR_SAL transaction field to be active. The MOVE keyword changes the value of CURR_SAL to that of the CURR_SAL field in the segment instance, even though the CURR_SAL field was already active. The transaction record is now:

```
Transaction Record:

EMP_ID: 451123478 (inactive)
CURR_SAL: 16100.00 (active)
DEPARTMENT: PRODUCTION (active)
ED_HRS: 50 (active)
```

12. The request branches to the NEXT_EMP3 case.
13. The request moves the current position to the next current instance after EMP_ID 451123478. This instance contains the following fields:

```
Database Segment Instance:

EMP_ID: 543729165
CURR_SAL: 9000.00
DEPARTMENT: MIS
ED_HRS: 25.00
```

14. The request updates the data source CURR_SAL, DEPARTMENT, and ED_HRS fields using the transaction record, causing the CURR_SAL, DEPARTMENT, and ED_HRS transaction fields to become inactive. The segment instance is now:

```
Database Segment Instance:

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

The transaction record is now:

```
Transaction Record:

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

Syntax: How to Deactivate Fields With the DEACTIVATE Statement

To deactivate a field, use the DEACTIVATE statement. If the field is a transaction field, the DEACTIVATE statement changes its value to blank if alphanumeric, zero if numeric, or the MISSING symbol for fields described by the MISSING=ON attribute (discussed in the *Describing Data* manual). It also deactivates the corresponding data source field. The RETAIN option leaves the transaction value unchanged.

The syntax is

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

where:

`RETAIN`

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

`field-1 ...`

Are the fields you want to deactivate. To deactivate all the fields in one segment, specify any segment field with the prefix seg. affixed in front of the field name. For example:

```
DEACTIVATE SEG.SKILLS
```

`ALL`

Is an option that deactivates all fields (including temporary fields) and automatically invokes the INVALID option if the request contains CRTFORM statements (see below).

`COMPUTES`

Is an option that deactivates all temporary fields.

`INVALID`

Is an option that causes the following: if the user enters a value on a CRTFORM screen and the value fails a validation test, FIDEL does not redisplay the CRTFORM screen to reprompt the user for a valid value. Rather, it displays the next screen.

Use the INVALID option only with requests containing CRTFORM statements.

The ACTIVATE and DEACTIVATE statements can stand by themselves or they can form part of an ON MATCH, ON NOMATCH, ON NEXT, or ON NONEXT phrase in a MATCH or NEXT statement. These are some sample statements:

```
ACTIVATE RETAIN SKILLS

ON MATCH DEACTIVATE ALL

ON NONEXT ACTIVATE FULL_NAME SEG.SKILLS JOBS_DONE
```

Protecting Against System Failures

FOCUS provides three ways to protect your data if your system experiences hardware or software failure while you are executing a MODIFY request. They are:

- The Checkpoint facility.
- The Absolute File Integrity feature.

- The COMMIT and ROLLBACK subcommands.

Syntax: How to Safeguard Transactions With the Checkpoint Facility

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

When a MODIFY request is executed, it does not write transactions to the data source immediately, instead it collects them in a buffer. When the buffer is full, FOCUS writes all transactions in the buffer to the data source at one time. This cuts down on the input/output operations that FOCUS must perform. If, however, the system crashes, the transactions collected in the buffer may be lost.

You may cause FOCUS to write more frequently to the data source by using the checkpoint facility. When you activate the Checkpoint facility, FOCUS writes to the data source whenever a specified number of transactions accumulates in the buffer. The point at which FOCUS writes the transactions is called the checkpoint.

You control the Checkpoint facility with the following MODIFY statement

```
CHECK {ON|OFF|n}
```

where:

`ON`

Activates the Checkpoint facility. FOCUS writes to the data source when the buffer accumulates 100,000 transactions.

`OFF`

Deactivates the Checkpoint facility.

n

Activates the Checkpoint facility. FOCUS writes to the data source when the buffer accumulates *n* transactions.

Note that if you set *n* to a smaller number, fewer transactions are processed between checkpoints. This causes FOCUS to perform more input/output operations, thereby decreasing efficiency.

If the system does fail while you are modifying a FOCUS data source, enter the ? FILE query when the system comes back. Look at the number in the bottom row in the right-most column. This is the number of transactions written to the data source by the MODIFY request that was executing when the system came down. You can have the request start processing the transaction data source at the next transaction by using the START command, described in *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* on page 73.

The following MODIFY request sets the checkpoint at every tenth transaction:

```
MODIFY FILE EMPLOYEE
CHECK 10
MATCH EMP_ID
PROMPT EMP_ID CURR_SAL
  ON MATCH UPDATE CURR_SAL
  ON NOMATCH REJECT
DATA
```

Reference: Safeguarding FOCUS Data Sources: Absolute File Integrity

The Absolute File Integrity feature completely safeguards the integrity of a FOCUS data source that you are modifying, even if the system experiences hardware or software failure. When you are using this feature, FOCUS does not overwrite the data source on disk, instead it writes the changes to another section of the disk. If the request finishes normally, the new section of the disk becomes part of the data source. If the system fails, the original data source is preserved.

Reference: Safeguarding Transactions: COMMIT and ROLLBACK Subcommands

To use COMMIT and ROLLBACK you must use Absolute File Integrity (see *Managing MODIFY Transactions: COMMIT and ROLLBACK* on page 218). Unlike the CHECK statement, COMMIT gives you control over the content of data source changes and ROLLBACK enables you to cancel changes before they have been written to the data source. In case of system failure, COMMIT and ROLLBACK ensure that either all or no transactions are processed.

You can use either COMMIT and ROLLBACK, or the CHECK statement in your MODIFY procedures. If the MODIFY procedure uses COMMIT and ROLLBACK, CHECK processing is not used (see *Managing MODIFY Transactions: COMMIT and ROLLBACK* on page 218).

Displaying MODIFY Request Logic: The ECHO Facility

The ECHO facility displays the logical structure of MODIFY requests. This is a good debugging tool for analyzing a MODIFY request, especially if the logic is complex and MATCH and NEXT defaults are being used.

Each ECHO display lists:

- The cases, if case logic is used.
- The MODIFY statements used, such as COMPUTE, VALIDATE, TYPE, GOTO, and IF.
- Each segment modified or used to establish a current position.
- The actions the request takes for ON MATCH, ON NOMATCH, ON NEXT, and ON NONEXT conditions when it is modifying the segment, whether these actions are specified by the request or are by default. Default actions are discussed in *The MATCH Statement* on page 75.
- The number of data source fields, the total number of fields (including internal fields), and the total size of the field areas.

To use the ECHO facility, first allocate the ECHO terminal output to ddname HLIPRINT. Then, begin the MODIFY command this way

```
MODIFY FILE file ECHO
```

where *file* is the name of the data source. When you run the request, the request does not modify the data source; rather, the ECHO facility displays the listing at the terminal.

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

The listing has the form

```
MODIFY ECHO FACILITY
ECHO OF PROCEDURE: focexec


------------------------------------------------------------------------
CASE casename
------------------------------------------------------------------------

statements

                  SEGMENT: segname

ON MATCH                ON NOMATCH
--------                ----------
match-actions             nomatch-actions0

NUMBER OF DATABASE FIELDS   : n
TOTAL NUMBER OF FIELDS      : n
TOTAL SIZE OF FIELD AREAS   : n
```

where:

focexec

Is the name of the procedure that the request is stored in. If you entered the request from a terminal, this line is omitted.

casename

Is the name of the case, if the request uses Case Logic.

statements

Are the MODIFY statements used. (**Note:** MATCH statements are shown separately.)

segname

Is the name of the segment being modified or used to establish a current position.

match-actions

Are actions taken on an ON MATCH or ON NEXT condition, including default actions.

nomatch-actions

Are actions taken on an ON NOMARCH or ON NONEXT condition, including default actions.

n

Is an integer.

`NUMBER OF DATABASE FIELDS`

Is the number of fields described by the Master File, including fields in cross-referenced segments.

`TOTAL NUMBER OF FIELDS`

Is the sum of the number of data source fields in the Master File and temporary fields in the MODIFY request. This includes fields automatically created by FOCUS (these fields are listed in *Computing Values: The COMPUTE Statement* on page 106).

`TOTAL SIZE OF FIELD AREAS`

Is the sum of the sizes of data source fields in the Master File and temporary fields in the MODIFY request, measured in bytes.

If you are executing a no-case procedure, the ECHO display lists the names of all segments in the data source. Those segments that you did not use in your request are listed with both MATCH and NOMATCH conditions as REJECT.

A sample request running the ECHO facility is shown below:

```
MODIFY FILE EMPLOYEE ECHO
PROMPT EMP_ID
GOTO SALENTRY

CASE SALENTRY
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 SALENTRY
  ON MATCH UPDATE CURR_SAL
ENDCASE
DATA
```

When you run this request, the following display appears. Note that although the request did not specify an ON NOMATCH phrase in the SALENTRY case, the ECHO display lists the REJECT action under the ON NOMATCH column for the SALENTRY case, because REJECT is the default action for an ON NOMATCH condition.

```
EMPLOYEE FOCUS A1 ON 07/18/2003 AT 10.48.21

    MODIFY ECHO FACILITY
    ECHO OF PROCEDURE: MOD76

----------------------------------------
CASE TOP
----------------------------------------
PROMPT
GOTO  SALENTRY

----------------------------------------
CASE SALENTRY
----------------------------------------
            SEGMENT: EMPINFO
         ------------------------
  MATCH                      NOMATCH
  -----                      -------
  PROMPT                     REJECT
  VALIDATE
  INVALID TYPE
  INVALID GOTO SALENTRY
  UPDATE

END OF ECHO:

 NUMBER OF DATABASE FIELDS : 34
 TOTAL NUMBER OF FIELDS    : 36
 TOTAL SIZE OF FIELD AREAS : 371
```

Dialogue Manager Statistical Variables

After you run a FOCUS request, FOCUS automatically records statistics about the execution in specially designated Dialogue Manager variables. Since these variables do not receive values until after execution is completed, they are not useful in the requests themselves. However, you may use them in procedures after execution (that is, after the Dialogue Manager -RUN control statement).

The variables that pertain to MODIFY requests are:

`&TRANS`	Number of transactions processed.
`&ACCEPTS`	Number of transactions accepted into the data source.
`&INPUT`	Number of segment instances added to the data source.
`&CHNGD`	Number of segment instances updated.
`&DELTD`	Number of segment instances deleted.
`&DUPLS`	Number of transactions rejected because of an ON MATCH REJECT condition.
`&NOMATCH`	Number of transactions rejected because of an ON NOMATCH REJECT condition.
`&INVALID`	Number of transactions rejected because transaction values failed validation tests.
`&FORMAT`	Number of transactions rejected because of format errors.
`&REJECT`	Number of transactions rejected for other reasons.

For instructions on how to use Dialogue Manager variables to build procedures, see the *Developing Applications* manual.

MODIFY Query Commands

Four query commands display information regarding the MODIFY command and the maintenance of FOCUS data sources. These are:

`? COMBINE`	Displays information on combined structures (see *Modifying Multiple Data Sources in One Request: The COMBINE Command* on page 196).
`? FDT`	Displays information regarding the physical attributes of FOCUS data sources (see the *Developing Applications* manual).
`? FILE`	Displays information regarding the number of segment instances in FOCUS data sources and the dates and times the data sources were last modified (see the *Developing Applications* manual).
`? STAT`	Displays statistics regarding the last execution of a request (see the *Developing Applications* manual).

Managing MODIFY Transactions: COMMIT and ROLLBACK

COMMIT and ROLLBACK are two MODIFY subcommands. COMMIT gives you control over the content of data source changes and ROLLBACK enables you to undo changes before they become permanent.

The COMMIT subcommand safeguards transactions in case of a system failure and provides greater control (than the MODIFY Checkpoint facility) over which transactions are written to the data source.

The MODIFY CHECK statement only enables you to control the number of transactions that must occur before changes are written to the data source. When using CHECK, you cannot change the checkpoint setting once the MODIFY request begins execution. Similarly, changes cannot be canceled (see *How to Safeguard Transactions With the Checkpoint Facility* on page 212 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 by the FOCUS Shadow Writing Facility.

Note: Absolute File Integrity is not supported for XFOCUS data sources and is not required for COMMIT and ROLLBACK.

Reference: The COMMIT and ROLLBACK Subcommands

The COMMIT and ROLLBACK subcommands are automatically activated in FOCUS and cannot be deactivated. Therefore, unless you omit these subcommands from your code, COMMIT and ROLLBACK processing takes place. If you would rather use CHECK processing, make sure you do not include COMMIT and ROLLBACK subcommands, as they will take precedence over CHECK processing.

Reference: Coding With COMMIT and ROLLBACK

COMMIT and ROLLBACK each process a logical transaction. A logical transaction is a group of data source changes in the MODIFY environment that you want to treat as one. For example, you can handle multiple records displayed on a CRTFORM and then processed using the REPEAT command as a single transaction. A logical transaction is terminated by either COMMIT or ROLLBACK. COMMIT and ROLLBACK also can be used for single-record processing.

When COMMIT ends a logical transaction, it writes all changes to the data source. COMMIT can be coded as a global subcommand or as part of MATCH or NEXT logic. The possible MATCH and NEXT statements are:

```
COMMIT
ON MATCH COMMIT
ON NOMATCH COMMIT
ON MATCH/NOMATCH COMMIT
ON NEXT COMMIT
ON NONEXT COMMIT
```

When ROLLBACK ends a logical transaction, it does not write changes to the data source. The ROLLBACK subcommand cancels changes made since the last COMMIT. ROLLBACK cannot cancel changes once a COMMIT has been issued for them.

ROLLBACK can also be coded as a global subcommand or as part of MATCH or NEXT logic. Possible MATCH and NEXT statements are:

```
ROLLBACK
ON MATCH ROLLBACK
ON NOMATCH ROLLBACK
ON MATCH/NOMATCH ROLLBACK
ON NEXT ROLLBACK
ON NONEXT ROLLBACK
```

If the COMMIT fails for any reason (for example, system failure, lack of disk space), no changes are made to the data source. In this way, COMMIT is an all-or-nothing feature that ensures data source integrity.

In the following example, a user may COMMIT or ROLLBACK changes after each group of three records has been processed, or delay the COMMIT subcommand until later by selecting the option to add more records. Changes are stored permanently in the data source when the user chooses to commit the changes or when the procedure is terminated without issuing a ROLLBACK subcommand.

Note: In the following example the COMMIT and ROLLBACK subcommands are included in Case COMM and Case ROLL, respectively.

```
MODIFY FILE EMPLOYEE
COMPUTE ANSWER/A1=;
CRTFORM LINE 1
"ENTER UP TO 3 NEW EMPLOYEES"
" "
"   EMPLOYEE ID   LAST NAME     FIRST NAME"
"1. <EMP_ID(1)   <LAST_NAME(1)  <FIRST_NAME(1)"
"2. <EMP_ID(2)   <LAST_NAME(2)  <FIRST_NAME(2)"
"3. <EMP_ID(3)   <LAST_NAME(3)  <FIRST_NAME(3)"
GOTO MATCHIT

CASE MATCHIT
REPEAT 3
  MATCH EMP_ID
    ON NOMATCH INCLUDE
    ON MATCH REJECT
ENDREPEAT
GOTO DECIDE
ENDCASE
```

```
CASE DECIDE
CRTFORM LINE 10
"WHAT WOULD YOU LIKE TO DO NOW? <ANSWER"
" C TO COMMIT CHANGES SO FAR"
" R TO ROLLBACK CHANGES"
" A TO ADD MORE EMPLOYEES"
IF ANSWER EQ 'C' PERFORM COMM
  ELSE IF ANSWER EQ 'R' PERFORM ROLL
  ELSE IF ANSWER EQ 'A' GOTO TOP
  ELSE PERFORM BADCHOICE;
GOTO TOP
ENDCASE

CASE COMM
COMMIT
ENDCASE

CASE ROLL
ROLLBACK
ENDCASE

CASE BADCHOICE
TYPE "PLEASE ENTER C, R, OR A."
GOTO DECIDE
ENDCASE

DATA
END
```

MODIFY Syntax Summary

This section presents a summary of MODIFY command syntax. The syntax of each statement is shown as part of a MODIFY request. The rest of the summary shows:

- The syntax of the transaction statements FIXFORM, FREEFORM, and PROMPT. The syntax of the CRTFORM statement is shown in *Designing Screens With FIDEL* on page 227.
- 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 [*|number] [TIMES] [MAX maximum] [NOHOLD]
  statements
  HOLD [SEG.]field [field ... field]
ENDREPEAT

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

DEACTIVATE {[RETAIN] [SEG.] field ... field |[RETAIN]
ALL|COMPUTES|INVALID}

CASE casename

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

PERFORM {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}

IF expression
[THEN] {GOTO|PERFORM} {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}
[ELSE {GOTO|PERFORM} {TOP|ENDCASE|ENDREPEAT|casename|variable|EXIT}]

HOLD [SEG.]field [field ... field]

GETHOLD

NEXT field
  ON NEXT action
  ON NEXT action
  .
  .
  ON NONEXT action
  ON NONEXT action
  .
  .
ENDCASE

COMMIT
ROLLBACK

LOG {TRANS|ACCEPTS|DUPL|NOMATCH|INVALID|FORMAT} [ON ddname]
[MSG {ON|OFF}]

CHECK {ON|OFF|n}

START n

STOP n

DATA {ON ddname|VIA progname}

[END]
```

Transaction Statement Syntax

The following is the syntax for three transaction statements: FIXFORM, FREEFORM, and PROMPT. For CRTFORM syntax, see *Designing Screens With FIDEL* on page 227.

The syntax of the FIXFORM statement:

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

The syntax of the FREEFORM statement:

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

The syntax of the PROMPT statement:

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

MATCH and NEXT Statement Actions

This section lists the actions that can be taken by MATCH and NEXT statements. They are placed in ON MATCH, ON NOMATCH, ON NEXT, and ON NONEXT phrases. These actions are:

- ACTIVATE
- COMMIT
- COMPUTE
- CONTINUE (ON MATCH and ON NEXT only)
- CONTINUE TO (ON MATCH and ON NEXT only)
- CRTFORM
- DEACTIVATE
- DELETE (ON MATCH and ON NEXT only)
- FIXFORM
- FREEFORM
- GOTO

- HOLD
- IF
- INCLUDE
- PERFORM
- PROMPT
- REJECT
- REPEAT (ON MATCH and ON NEXT only)
- ROLLBACK
- TED (ON MATCH and ON NOMATCH ON NEXT and ON NONEXT
- TYPE
- UPDATE (ON MATCH and ON NEXT only)
- VALIDATE

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

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

The following actions can be used in ON INVALID phrases:

```
GOTO
PERFORM
TYPE
```

Chapter 2

Designing Screens With FIDEL

FIDEL, the FOCUS Interactive Data Entry Language, enables you to design full-screen forms for data entry and application development. You use FIDEL both with MODIFY for building data maintenance and inquiry screens, and with Dialogue Manager for building applications that accept values for variables at run time.

In this chapter:

Introduction

Describing the CRT Screen on page 232 describes the facilities of FIDEL that are common to both MODIFY and Dialogue Manager. This introduction explains how MODIFY facilities and FIDEL interact, and describes the FIDEL facilities that are specific to MODIFY. *Using FIDEL in Dialogue Manager* on page 297 describes the interaction between Dialogue Manager and FIDEL.

From the FOCUS TED editor, you can also use the FOCUS Screen Painter with both MODIFY and Dialogue Manager to interactively build and view screens online. With the Screen Painter, you design the layout of the form and the Screen Painter automatically generates the FIDEL code to build it. The FOCUS Screen Painter is described in *Using the FOCUS Screen Painter* on page 302.

The two simple examples on the following pages demonstrate how to generate a screen form by using the CRTFORM and -CRTFORM syntax. Note how closely FIDEL syntax resembles TABLE syntax for creating headings.

Note: FIDEL only supports fixed format records with LRECL=80.

Using FIDEL With MODIFY

The following example of a simple MODIFY CRTFORM illustrates the use of FIDEL with the resulting screen (the numbers refer to the explanation and are *not* part of the code):

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
2.     "EMPLOYEE UPDATE"
3.     "EMPLOYEE ID #: <EMP_ID   LAST NAME:  <LAST_NAME"
4.     "DEPARTMENT: <DEPARTMENT  SALARY:     <CURR_SAL"


5.  MATCH EMP_ID
       ON NOMATCH REJECT
       ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
6.  DATA
    END
```

This request sets up a form to update the last name, department and current salary. Processing is as follows:

1. CRTFORM generates the visual form and invokes FIDEL. The form begins on line one of the screen unless specified otherwise with the LINE option (see *Using Multiple CRTFORMs: LINE* on page 274).
2. Each line on the screen begins and ends with double quotation marks. This is a line of text that serves as a title. Note the close correspondence to the syntax used to create headings in a TABLE request.
3. The second screen line specifies two data fields: EMP_ID and LAST_NAME. A data entry field is indicated by a left caret, followed by the field name or alias from the Master File. The text, EMPLOYEE ID #: and LAST NAME: identifies each field on the screen. This informs the operator where to enter the data.
4. This is the last line within double quotation marks. It signals the end of the CRTFORM. In this case it identifies and defines two more data fields: DEPARTMENT and CURR_SAL. When you run the MODIFY request, the form instantly appears on the screen:

   ```
   EMPLOYEE UPDATE
   EMPLOYEE ID #:  LAST NAME:
   DEPARTMENT:  SALARY:
   ```

   The number of characters allotted for each data entry field on the screen defaults to the display format for that particular field in the Master File. You can optionally specify a format for screen display that is shorter than the default.

   The operator can now fill in the data entry areas with the appropriate information.
5. The request continues with MODIFY MATCH logic.

6. This line tells FOCUS that the incoming data is from the terminal. In conjunction with CRTFORM, it implies full-screen data input. You can also use DATA VIA FIDEL.

When you use FIDEL with MODIFY, you are setting up full-screen forms for the maintenance of data source fields. Most MODIFY features, such as conditional and non-conditional fields, automatic application generation, Case Logic, multiple record processing, error handling, validation tests, logging transactions, and typing messages to the terminal, work with FIDEL.

With MODIFY you also have access to additional screen control options such as clearing the screen, specifying and changing the size of the screen, and designating the particular line on which the form starts.

Using FIDEL With Dialogue Manager

The following example of a simple -CRTFORM illustrates the use of FIDEL in Dialogue Manager and the resulting screen (the numbers refer to the explanation and are *not* part of the code):

```
1. -CRTFORM
2. -"MONTHLY SALES REPORT FOR <&CITY/10"
3. -"BEGINNING PRODUCT CODE IS: <&CODE1/3"
   -"ENDING PRODUCT CODE IS: <&CODE2/3"
4. -"REGIONAL SUPERVISOR IS: <&REGIONMGR/5"
   TABLE FILE SALES
   HEADING CENTER
   "MONTHLY REPORT FOR &CITY"
   "PRODUCT CODES FROM &CODE1 TO &CODE2"
   " "
   SUM UNIT_SOLD AND RETURNS AND COMPUTE
   RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
   BY PROD_CODE
   IF PROD_CODE IS-FROM &CODE1 TO &CODE2
   FOOTING CENTER
   "REGIONAL SUPERVISOR: &REGIONMGR"
   END
```

The procedure sets up a form for gathering run-time variables for a TABLE request: &CITY, the city for the report; &CODE1 and &CODE2, a range of product codes; and &REGIONMGR, the regional supervisor. Processing is as follows:

1. -CRTFORM generates the visual form, invokes FIDEL, and clears the screen.
2. Each line on the screen begins with a dash and double quotation marks (-"), and ends with double quotation marks. Note this first line of the screen form contains text and a variable field, &CITY, which has a length of 10. This specifies ten spaces on the screen for entering the value. The data entry field is indicated by the left caret.
3. The next few lines of the screen form contain both text and variable fields with formats.

4. The last line within double quotation marks signals the end of the -CRTFORM. When the FOCEXEC executes, the screen displays the following form:

```
MONTHLY SALES REPORT FOR
BEGINNING PRODUCT CODE IS:
ENDING PRODUCT CODE IS:
REGIONAL SUPERVISOR IS:
```

   The operator can now fill in values for the run-time variables. After the operator transmits the screen by pressing Enter, the values entered on the screen are sent to the variables. The regular FOCUS commands are stacked and executed when the end of the procedure is reached.

When you use FIDEL with Dialogue Manager, you can define input fields as amper variables that receive values at run time to adjust to specific processing requirements. Because they are *not* data fields and are not part of the Master File, they do not automatically have a format. You must allocate space for them on the screen. You can do this directly on the -CRTFORM as in the previous example, or through a -SET statement.

Dialogue Manager supports two additional control statements: -CRTFORM BEGIN and -CRTFORM END. The statement -CRTFORM BEGIN signals the beginning of the screen form. You can then enter screen lines as well as other Dialogue Manager control statements. You then signal the end of the screen form with the statement -CRTFORM END. This allows you to use Dialogue Manager statements between screen lines while building the form.

Screen Management Concepts and Facilities

The following briefly outlines the FIDEL capabilities that are common to both MODIFY and Dialogue Manager and defines the common terminology:

- The MODIFY CRTFORM statement and the Dialogue Manager -CRTFORM control statement both automatically invoke FIDEL. All succeeding lines placed within double quotations make up the actual screen form. Note the common syntax between TABLE headings (see the *Creating Reports* manual) and CRTFORM screen lines.
- You can combine a CRTFORM and a -CRTFORM in one procedure. However, they must remain within their own environments. The MODIFY CRTFORM contains data source fields, whereas the Dialogue Manager -CRTFORM contains amper variables.
- The term *field* in this chapter refers to either a data source field name in conjunction with MODIFY or an amper variable in conjunction with Dialogue Manager.
- You can define a CRTFORM in MODIFY or a -CRTFORM in Dialogue Manager that has more lines than on your CRT screen. FIDEL provides scrolling capabilities.

- It is important to note the difference between the physical screen on the terminal and the logical CRTFORM or form. A form generated by one CRTFORM or -CRTFORM statement can take up many screens or less than one screen.
- You can specify three types of fields on the screen: input, display only, and turnaround (both display and update). Data entry and turnaround fields are considered unprotected areas on the screen because you may input values or replace what is there. Display values are considered protected areas on the screen because you cannot alter what is there (see *Data Entry, Display and Turnaround Fields* on page 239).
- You can set PF key controls and specify cursor positioning. You can specify screen attributes such as background effects, highlighting, and color to enhance readability of the screen. You can also change screen attributes depending on the outcome of various tests (see *Controlling the Use of PF Keys* on page 244, *Specifying Screen Attributes* on page 248, and *Using Labeled Fields* on page 252).

Note: This chapter is written specifically for the IBM 3270 terminal, which supports PF key and cursor control, scrolling and screen attributes.

Using FIDEL Screens: Operating Conventions

The following procedures apply for filling in *all* FIDEL screens:

- To move from field to field, press the Tab key. You can also move the cursor around the screen using the arrow keys.
- When filling in values on the screen, you may use any of the keys on the keyboard. Some terminals automatically prevent the entry of a non-numeric character in a field identified as computational.
- To scroll forward or backward through a long CRTFORM (from screen to screen) press the PF8 or PF7 key, respectively (or PF20, PF19).
- To transmit the screen, press the Enter key.
- If you make an error, the transaction may not be transmitted and an error message may appear at the bottom of the screen. You can correct the error and retransmit the screen.
- To signal the end of data entry, press the PF3 or PF15 key or type END in an unprotected area. In MODIFY, this terminates the request. In Dialogue Manager, this terminates the FOCEXEC procedure.

The following operating procedures are specific to MODIFY:

- To return to the first screen without transmitting the current screen, press the PF2 key or the key set to QUIT.

- ❑ If the screen clears at any time, press the Enter key to bring it back.

Note: The PF key settings referred to here are the default settings. Any PF key can be redefined using the SET statement.

Describing the CRT Screen

The MODIFY statement CRTFORM or the Dialogue Manager control statement -CRTFORM, followed by the screen layout, generates a form. Within one MODIFY procedure, you can use an unlimited number of screen lines (within memory constraints). Each screen line can contain a maximum of 78 characters of text and data.

In MODIFY, you can use up to 255 CRTFORM statements in a procedure. In Dialogue Manager, there is no limit to the number of -CRTFORM statements that you may use in one procedure.

All the basic options described here can be used with both MODIFY and Dialogue Manager. Options that are specific to MODIFY are discussed in *Using FIDEL in MODIFY* on page 264 and those specific to Dialogue Manager are discussed in *Using FIDEL in Dialogue Manager* on page 297.

The following example shows the syntax of a simple MODIFY CRTFORM using the LOWER case option, followed by two screen lines containing various screen elements: text, a spot marker, and a field (numbers refer to the explanation; they are *not* part of the code):

```
1. CRTFORM LOWER
2. "PLEASE FILL IN THE EMPLOYEE ID # </1"
3. "EMPLOYEE ID #: <EMP_ID"
   MATCH EMP_ID
     .
     .
     .
```

Processing is as follows:

1. CRTFORM invokes FIDEL and generates the form. The LOWER case option specifies that what is entered from the terminal in lowercase will remain in lowercase.
2. The first line of the screen contains descriptive text.

   </1 is a spot marker which skips one blank line.
3. The last line of the screen contains two screen elements: descriptive text that identifies the field and the data source field EMP_ID. The last line between quotation marks signals the end of the CRTFORM.

The form generated appears as follows:

```
PLEASE FILL IN THE EMPLOYEE ID #
```

```
EMPLOYEE ID #:
```

Specifying Elements of the CRTFORM

To create the visual form, you enter the screen lines one after the other within double quotation marks. For each screen line, you can specify various screen elements such as descriptive text and fields. A left caret (<) followed by the name of the field generates the position where data is to be entered onto the screen.

You may need to use two FOCEXEC lines to describe one physical CRTFORM line. Simply omit the double quotation marks (") at the end of the first line and omit them at the beginning of the next line as well. Everything between the set of double quotation marks will read as one screen line on the CRTFORM.

Syntax: How to Invoking FIDEL: CRTFORM and -CRTFORM

The following is a summary of the complete syntax for generating a CRTFORM in MODIFY or a -CRTFORM in Dialogue Manager. The individual options and screen elements are described in detail in specific sections later in the chapter. The syntax is

```
[-]CRTFORM [option option...]
[-]"screen element [screen element....]"
```

where:

`[-]CRTFORM`

Automatically invokes FIDEL and sets up the visual form. Subsequent lines describe the screen.

`option option...`

Refers to screen control options. (See *Using FIDEL in MODIFY* on page 264 and *Using FIDEL in Dialogue Manager* on page 297.)

`[-]"`*`screen element..`*`"`

Can be user-defined text, fields, or spot markers. Spot markers define the next place on the screen where a screen element will appear. Both spot markers and fields are preceded by a left caret and optionally closed by a right caret (see *Specifying Elements of the CRTFORM* on page 233).

Note:

- You can create simple screen forms by typing the FIDEL code into your procedures with your text editor. However, it is easier to build more complex forms using many screen attributes and field labels using the FOCUS Screen Painter.

- You can use the asterisk (*) with CRTFORM in FIDEL to generate a CRTFORM containing all of the data source's fields automatically (that is, without specifying individual fields). See *Generating Automatic CRTFORMs* on page 270 for information on CRTFORM *, its syntax and variations.
- Do not begin any field used in a CRTFORM or FIXFORM statement with X*n*, where *n* is any numeric value. This applies to fields in the Master File and computed fields.

Defining a Field

Labels, prefixes, attributes, and formats are parts of the definition of a particular field. In Dialogue Manager, the first character is an ampersand, which signals an amper variable. (The entire definition is preceded by a left caret and optionally closed by a right caret.)

Note: Fields with a text (TX) format cannot be used in CRTFORM or -CRTFORM. However, they can be entered interactively using TED (see *Entering Text Data Using TED* on page 69, for using text fields in MODIFY).

Syntax: How to Define a Field in FIDEL

The syntax for defining a field is as follows.

In MODIFY:

```
<[:label.][prefix.][attribute.]field[/length][>]
```

In Dialogue Manager:

```
<[&:label.][prefix.][attribute.]&variable[/length][>]
```

where:

`:label.|&:label.`

Is a user-defined label of up to 12 characters associated with a field. It may not contain embedded blanks (see *Using Labeled Fields* on page 252).

prefix.

Refers to D. or T., which designate a display or turnaround field, respectively (see *Data Entry, Display and Turnaround Fields* on page 239).

attribute.

Is the abbreviation or full name of a screen attribute (see *Specifying Screen Attributes* on page 248).

field

Is the name of the field or variable being defined.

&variable

Is for data entry. Can be a data source field or a temporary field.

/length

Is the length of the field as it appears on the screen. In MODIFY, you need to define a length only if you want the screen length to be different from the format length that is defined in the MASTER or COMPUTE. In Dialogue Manager, you need to define a length only if not previously defined.

Note: When you use the abbreviations for attributes, you do not need to use the dot separator between attributes or between a prefix and an attribute (see *Specifying Screen Attributes* on page 248).

Example: **Defining a Field**

The following is an example of the syntax of a Dialogue Manager screen line defining the variable field &CITY:

```
-CRTFORM
-"<&:L01.T.HIGH.&CITY/7"
        .
        .
        .
```

The elements on the second line which define the variable field &CITY are:

1. The left caret generates a place for the variable on the screen.
2. &:L01 is a label that identifies the data entry area on the screen (see *Using Labeled Fields* on page 252).
3. T. is a prefix that defines the variable as a turnaround field. If the variable has been given a value within the FOCEXEC, it is displayed. Otherwise a default value is displayed. The operator can then change the value.
4. .HIGH. is a screen attribute specifying that the contents of the field will be highlighted.
5. &CITY/7 is the name of the variable field with a length specification. The specified length is seven characters. That is, the space that will be allotted on the screen for input of data is seven characters long.

Prefixes, labels, and screen attributes are explained fully in *Data Entry, Display and Turnaround Fields* on page 239, *Specifying Screen Attributes* on page 248, and *Using Labeled Fields* on page 252.

Reference: Difference in FIDEL When Used With MODIFY and Dialogue Manager

The following chart outlines the similarities and differences of FIDEL when used with MODIFY and Dialogue Manager:

MODIFY	Dialogue Manager
CRTFORM [*options*]	-CRTFORM [*options*]
UPPER/LOWER CLEAR/NOCLEAR WIDTH/HEIGHT TYPE LINE	UPPER/LOWER BEGIN/END TYPE
"*screen elements*" *text* <*spot marker*[>]** <*field/length*[>]* *prefix.*(D. or T.)*** *attribute.* :*label.*	"*screen elements*" *text* <*spot marker*[>]** <*field/length*[>]** *prefix.*(D. or T.)*** *attribute* &:*label.*

* The right caret denotes a non-conditional field.

** The right caret has no meaning, but may be used for increased clarity.

*** Prefixes, attributes and labels are part of the definition of the field on the screen. They do not stand alone.

Using Spot Markers for Text and Field Positioning

Because the lengths of fields vary, text does not automatically align uniformly on the screen. Spot markers are available to help you position both text and fields. Please note that a spot marker is essential to eliminate trailing blanks at the end of the first line, if your screen line description takes up two FOCEXEC lines.

The syntax and usage of the different spot markers are shown in the following chart:

Marker	Example	Usage
<*n* or <*n*>	<50	Positions the next character in column 50.

Marker	Example	Usage
<+*n* or <+*n*>	<+4	Positions the next character four columns from the last non-blank character.
<-*n* or <-*n*>	<-1	Positions the next character one column to the left of the last character. This marker's function is to suppress or write over the attribute byte at the beginning and the end of a field.
</*n* or </*n*>	</2	Positions the next character at the beginning of the line that is two lines from the last (skips two lines). **Note:** The last line is blank and is created when a double quotation mark (") is encountered.
<0X or <0X>	<0X	Positions the next character immediately to the right of the last character (skip zero columns). This is used to help position data on a FIDEL screen when a single screen line is coded as two lines in a FOCEXEC. No spaces are inserted between the spot marker and the start of a continuation line (see Note 3 in the following example).

Note: You can optionally use the right caret >. This is useful when the next character in the line is a left caret. It also enhances readability.

Suppose you want the various input data fields arranged across the screen in vertical sections, left justified, and in horizontal segments marked off with lines. Using spot markers, you can create the desired screen as shown in the following example:

```
   MODIFY FILE EMPLOYEE
   CRTFORM
    "EMPLOYEE UPDATE"
1.  "</1"
    "------------------------------------------------------------"
    "EMPLOYEE ID #: <EMP_ID   LAST NAME: <LAST_NAME"
1.  "</1"
2.  "DEPARTMENT: <DEPARTMENT <+3 CURRENT SALARY:<0X>
    <CURR_SAL"
    "------------------------------------------------------------"
    "BANK: <BANK_NAME"
    "------------------------------------------------------------"
   MATCH EMP_ID
   .
   .
   .
   DATA
   END
```

The spot markers in the example perform the following functions:

1. </1 generates a blank line.
2. <+3 moves the word CURRENT three spaces to the right of the last letter in the word DEPARTMENT. <0X> skips no spaces. No extra spaces are inserted between this and the next word (<CURR_SAL) on the continuation line. There is, in fact, one space before the field which is an attribute byte that marks the start of a field.

The screen appears as:

```
EMPLOYEE UPDATE


----------------------------------------
EMPLOYEE ID #:  LAST NAME:


DEPARTMENT:  CURRENT SALARY:
----------------------------------------
BANK:
----------------------------------------
```

Specifying Lowercase Entry: UPPER/LOWER

All text that is entered from the terminal is normally translated to uppercase letters. You can override this default and preserve both uppercase and lowercase text by using the lowercase option. The syntax is

```
[-]CRTFORM [UPPER|LOWER]
```

where:

`UPPER`

Translates all characters to uppercase. This is the default.

`LOWER`

Reads lowercase data from the screen. Once you specify LOWER, every screen thereafter is a lowercase screen until you specify UPPER.

Note: In MODIFY, when you use multiple CRTFORMs on the same screen (using LINE n), you can mix UPPER and LOWER among the forms.

Data Entry, Display and Turnaround Fields

There are three types of data or variable fields that can be specified on the CRTFORM: data entry, display, and turnaround.

You can also compute data fields (see *Computing Values: The COMPUTE Statement* on page 106, for rules about computing data fields) and specify them as entry, display, or turnaround on the CRTFORM. You can convert a turnaround field to a display field dynamically.

In MODIFY, fields can also be designated as conditional or unconditional (see *Conditional and Non-Conditional Fields* on page 264). We recommend that for data entry, you use conditional fields (left caret only) so that the values in your data source are not replaced by a blank or a zero if you do not enter data for the field.

For most turnaround fields, we recommend that you use non-conditional fields (both carets). A non-conditional turnaround field remains active whether you enter data or not. Because the value in the data source is displayed in the field, that value remains in the data source if you do not change it. Because the field remains active, the values for your VALIDATEs and COMPUTEs are then accurate (see *Conditional and Non-Conditional Fields* on page 264 for a complete explanation of the use of conditional and non-conditional fields in MODIFY).

The following outlines the rules for specification of different types of fields.

Syntax: How to Use Data Entry Fields (for Data Entry Only)

In MODIFY, the syntax is

<field[*/length*][>]

where:

<field[>]

Is the name of the field. Reserves space on the screen for data entry into that field and does not display the current value of the field.

In MODIFY, if only the left caret is used, data entry is conditional. If both carets are used, the field is non-conditional (see *Conditional and Non-Conditional Fields* on page 264).

In Dialogue Manager the syntax is

```
<&variable[/length][>]
```

where:

```
<&variable[>]
```

Is the name of the variable field. Reserves space on the screen for data entry into that field and does not display the current value of the field.

In Dialogue Manager, the option of the right caret is meaningless. Usually for the FOCEXEC to run, you must supply a value for each variable. If you do not, FOCUS assumes a blank or a 0 for that value.

Syntax: How to Use Display Fields (for Information Only)

Data is displayed in a protected area and cannot be altered.

In MODIFY, the syntax is

```
<D.field[/length]
```

In Dialogue Manager, the syntax is

```
<D.&variable[/length]
```

where:

```
D.
```

Is the prefix placed in front of a field, indicating that the data or value is to be displayed. The current value of the field appears on the screen, but in a protected area which cannot be changed.

Note that the right caret is meaningless for display fields.

Syntax: How to Use Turnaround Fields (for Display and Change)

Data is displayed in an unprotected area and can be altered.

In MODIFY, the syntax is:

```
<T.field[/length][>]
```

In Dialogue Manager, the syntax is:

```
<T.&variable[/length][>]
```

where:

`T.`

Is the prefix placed in front of a field to indicate that it is a turnaround field. The current value of the field is displayed on the screen. However, the operator may change the value, as it is not in a protected area.

In MODIFY, if only the left caret is present, the T. field is treated as conditional. If the right caret is used, the field is non-conditional, and the value is treated as present, even if unchanged (see *Conditional and Non-Conditional Fields* on page 264).

In Dialogue Manager, the changed value for the turnaround variable field will substitute everywhere in the FOCEXEC where it is subsequently encountered.

Note: In MODIFY, in order to display data from a data source field or present it for turnaround, a position in the data source must first be established through the use of a MATCH or NEXT statement, or value must be assigned in a COMPUTE. A computed field cannot be set and displayed in the TOP case, where data entry is processed prior to computations. For example, one of the phrases

```
ON MATCH CRTFORM
ON NEXT CRTFORM
```

must be used. A position is thus established in the data source, and the values of the fields in existing records are now available for display as protected or unprotected fields.

You can also match on a key field and go to a case (see *CRTFORMs and Case Logic* on page 279) in which you display a CRTFORM using display and turnaround fields.

Using Data Entry, Display, and Turnaround Fields

This section will show how to use Date Entry, Display, and Turnaround Fields with MODIFY and Dialogue Manager.

Example: Using Data Entry, Display, and Turnaround Fields With MODIFY

The following example combines two CRTFORMs in a single MODIFY request and shows the use of entry, display and turnaround fields (numbers refer to the explanation below; they are *not* part of the code):

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
       "ENTER EMPLOYEE ID#: <EMP_ID"
       "PRESS ENTER"
       "</2"
2.  MATCH EMP_ID
       ON NOMATCH REJECT
       ON MATCH CRTFORM
          " "
          "REVISE DATA FOR SALARY AND DEPARTMENT"
          "ENTER NEW DATA FOR EDUCATION HOURS"
          " "
3.        "EMPLOYEE ID #: <D.EMP_ID   LAST_NAME: <D.LAST_NAME"
          " "
4.        "SALARY:    <T.CURR_SAL>"
          "DEPARTMENT: <T.DEPARTMENT>"
5.        "EDUCATION HOURS: <ED_HRS>"
       ON MATCH UPDATE CURR_SAL DEPARTMENT ED_HRS
    DATA
    END
```

The procedure matches the employee ID, displays both the ID and the last name, and then displays the current salary and department for turnaround. Education hours is a data entry field.

Note that when the procedure executes, both CRTFORMs are displayed immediately. However, the display and turnaround fields in the second CRTFORM do not display data until the operator fills in the first form and presses Enter. We therefore recommend you use the LINE option.

When a FORMAT ERROR occurs, all data entered up to that point is processed and cannot be changed in the course of your transaction.

The processing is as follows:

1. CRTFORM generates the first form which begins on line 1 (the second CRTFORM is displayed, but without values):

```
ENTER EMPLOYEE ID #:
PRESS ENTER

REVISE DATA FOR SALARY AND DEPARTMENT
ENTER NEW DATA FOR EDUCATION HOURS

EMPLOYEE ID #:   LAST NAME:
SALARY:
DEPARTMENT:
EDUCATION HOURS:
```

2. The procedure continues with the MATCH logic. If the ID number that is input matches an ID in the data source, the display and turnaround fields on the second CRTFORM display the data. Assume the operator enters 818692173 and presses Enter.

   The following is displayed:

```
ENTER EMPLOYEE ID #: 818692173
PRESS ENTER

REVISE DATA FOR SALARY AND DEPARTMENT
ENTER NEW DATA FOR EDUCATION HOURS

EMPLOYEE ID #: 818692173     LAST NAME: CROSS
SALARY:   27062.00
DEPARTMENT:  MIS
EDUCATION HOURS:
```

3. This screen line contains two display fields.
4. The next two screen lines contain turnaround fields.
5. The last line is a data entry field.

Note: To display fields from a unique segment, the ON MATCH CONTINUE TO, ON NEXT, or MATCH WITH-UNIQUES phrase must have been executed (see *Modifying Data: MATCH and NEXT* on page 75).

In Dialogue Manager, in order to display values with D. or T., a value must have been supplied for the variable prior to the initiation of the -CRTFORM. System variables are an exception to this rule, as the system automatically supplies their values.

Computed fields in both MODIFY and Dialogue Manager can be displayed in any kind of CRTFORM.

Example: Using Data Entry, Display, and Turnaround Fields With Dialogue Manager

The following example illustrates the use of D. fields and system variables in a Dialogue Manager -CRTFORM:

```
1.  -SET &CITY = STAMFORD;

2.  -CRTFORM
3.  -"YEARLY SALES REPORT FOR <T.&CITY/10"
4.  -"DATE: <D.&DATE  TIME: <D.&DATEMDYY"
    -" "
    -"ENTER BEGINNING PRODUCT CODE RANGE: <&BEGCODE/3"
    -"ENTER ENDING PRODUCT CODE RANGE: <&ENDCODE/3"
    -"ENTER NAME OF REGIONAL SUPERVISOR: <&REGIONMGR/15"
```

```
TABLE FILE SALES

HEADING CENTER
"YEARLY REPORT FOR &CITY"
"PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
" "
SUM UNIT_SOLD AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
IF CITY EQ &CITY
FOOTING CENTER
"REGION MANAGER: &REGIONMGR"
"CALCULATED AS OF &DATE"
END
```

The example processes as follows:

1. The -SET sets a default value for &CITY:

```
 FOR WHICH CITY DO YOU WANT A REPORT?
```

2. -CRTFORM generates the screen form:

```
YEARLY SALES REPORT FOR STAMFORD
DATE: 02/22/2003    TIME: 13.42.38

ENTER BEGINNING PRODUCT CODE RANGE:
ENTER ENDING PRODUCT CODE RANGE:
ENTER NAME OF REGIONAL SUPERVISOR:
```

3. The transaction value for &CITY is Stamford, the value that was previously supplied in the -SET statement.
4. Note that the variables &DATE and &DATEMDYY are system variables. The values are supplied by the system and displayed on the form.

Controlling the Use of PF Keys

The terminal operator can use certain PF keys to control the execution of a FIDEL application. Normally, the following keys are used:

- PF3 and PF15 mean END and terminate execution.
- PF2 means Cancel and cancels the transaction in MODIFY.
- PF7 and PF8 page Backward and Forward respectively.

Note: All other keys return the value of the PF key when pressed.

Several facilities are available to assist you in controlling various screen operations:

- You can reset PF key functions. You can also set PF keys to branch to particular cases in MODIFY or labels in Dialogue Manager.
- You can set the cursor on a specified position on the screen (see *Specifying Cursor Position* on page 256).
- You can use the cursor position on the screen to perform a branch or action based on a test (see *Determining Current Cursor Position for Branching Purposes* on page 258).

Reference: **Default Settings for PF Keys**

The default PF key settings are as follows:

PF Key	Function
PF01	HX
PF02	CANCEL
PF03, PF15	END
PF04, PF16	RETURN
PF05, PF17	RETURN
PF06, PF18	RETURN
PF07, PF19	BACKWARD
PF08, PF20	FORWARD
PF09, PF21	RETURN
PF10, PF22	RETURN
PF11, PF23	RETURN
PF13	RETURN
PF12, PF24	UNDO

PF Key	Function
PF14	RETURN

You can display the current PF key settings by issuing the FOCUS query command:

```
? PFKEY
```

This displays a formatted table of all the current values.

Resetting PF Key Controls

You can reset PF key functions in FIDEL for both CRTFORMs and -CRTFORMs using the FOCUS SET command with the following syntax

```
SET PFxx = function
```

where:

xx

Is a one or two-digit PF key number.

function

Is one of the following:

END in MODIFY, exits the procedure; in Dialogue Manager, is equivalent to QUIT. That is, END exits the procedure.

CANCEL in MODIFY, cancels the transaction and returns to the TOP case. Do not use the CANCEL setting in Dialogue Manager.

FORWARD pages forward.

BACKWARD pages backward.

RETURN has no specific screen action. Returns the PF key name in the PFKEY field because it is not yet defined. To set the PFKEY field, use COMPUTE in MODIFY or -SET in Dialogue Manager.

HELP displays text supplied with the HELPMESSAGE attribute for any field on the MODIFY CRTFORM. Position the cursor on the data entry area of the desired field, and press the PF key you have defined for HELP. If no help message exists for that field, the following message is displayed:

```
NO HELP AVAILABLE FOR THIS FIELD.
```

The following example sets the PF03 key for paging backward and the PF04 key for paging forward:

```
SET PF03=BACKWARD,PF04=FORWARD
```

Note: When changing PF key settings, make sure that at least one key is set to END. If you set a PF key to FORWARD, you should also set one to BACKWARD.

Setting PF Key Fields for Branching Purposes

You can create a menu of processing options. The operator can then indicate a choice by pressing a particular PF key. To assign a specific processing function to a PF key, you must specify a field named PFKEY. Which PF key the operator presses determines the value of the PFKEY field.

You can use the PF keys designated as Return keys, as well as the Enter key. You define a variable called PFKEY (in MODIFY) or &PFKEY (in Dialogue Manager) and then test its value after the CRTFORM is displayed. Which branch takes place depends on which PFKEY the operator presses.

In MODIFY, the syntax is

```
COMPUTE
PFKEY/A4=;
```

where:

`PFKEY/A4`

Is a four-character field, whose value is determined by which key the operator presses at run time.

In Dialogue Manager, the syntax is

```
-SET &PFKEY='  ';
```

where:

`&PFKEY`

Is a four-character field, whose value is determined by which key the operator presses at run time.

`=' ';`

Is the allocation of four character spaces for the field.

The following example shows how PF keys can be tested in MODIFY:

```
1.  COMPUTE
    PFKEY/A4=;
2.  CRTFORM
       "SELECT OPTION"
       "INPUT   PRESS PF4"
       "UPDATE  PRESS PF5"
       "DELETE   PRESS PF6"
3.  IF PFKEY EQ 'PF04' GOTO INCASE
       ELSE IF PFKEY EQ 'PF05' GOTO UPCASE
       ELSE IF PFKEY EQ 'PF06' GOTO DELCASE
       ELSE GOTO TOP;
            .
            .
            .
```

The example processes as follows:

1. The COMPUTE statement specifies a four-character field PFKEY.
2. CRTFORM generates the form which supplies the operator with three options:

```
SELECT OPTION
INPUT   PRESS PF4
UPDATE  PRESS PF5
DELETE  PRESS PF6
```

3. The IF test determines what case to branch to depending on the value of the PFKEY field. For example, if the operator presses PF4, the value for PFKEY is PF04, and the request branches to an input case INCASE.

Specifying Screen Attributes

Screen attributes (such as highlighting, colors, and so on) can be applied to the fields on the CRTFORM and the -CRTFORM. They can also be used as background effects and can be applied to the fields depending on the result of tests.

The following attributes are available on 3270 IBM terminals:

Function	Abbreviation	Short Name
Flash or Blink	F	FLAS or BLIN
Underline	U	UNDE
Invert or Reverse Video	I	INVE or REVV

Function	Abbreviation	Short Name
Clear*	C	CLEA
Blue	B	BLUE
Red	R	RED
Pink	P	PINK
Green	G	GREE
Aqua	A	AQUA
Turquoise	T	TURQ
Yellow	Y	YELL
White	W	WHIT
Nodisplay*	N	NODI
Return to default	$	$
Highlight or Intensify*	H	HIGH or INTE

Note:

- *Clear, Nodisplay, and Highlight or Intensify can be used on all terminals. Clear also sets the highlight off for entry and turnaround fields. Nodisplay is not supported for D. or T. fields. The remaining attributes are also known in the FOCUS community as extended attributes.
- Use of abbreviations is recommended, except for TURQ.

When an attribute is unsupported on a particular terminal or is specific to a version of FOCUS under another operating system, the attribute is ignored. Therefore, there is no need for code changes between terminals and/or operating systems.

To use the screen attributes other than C, N, and H you must notify FOCUS that your terminal is equipped to display them. Issue the FOCUS SET command:

```
SET EXTTERM=ON
```

This allows a procedure to be operated on a variety of terminals. FOCUS automatically detects a 3279 model terminal and sets EXTTERM to ON by default.

If your terminal does not properly recognize extended attributes, due to a "terminfo" compatibility problem, stray characters will appear on your screen. You may turn off extended attribute recognition with the command:

```
SET EXTTERM=OFF
```

Programs with extended attributes and EXTTERM=OFF will run as if extended attributes had not been coded in the program.

Make sure that your terminal has the extended attribute options needed before you turn EXTTERM on. There are many different IBM 3270 models. Generally, the color terminals in the 3279 series have most of the options. However, even if a terminal has the physical capability to support all of the attributes, it may be defined to the operating system as a lower grade terminal. In such cases, you must ascertain whether or not all the attributes can be used.

The syntax for defining screen attributes in MODIFY is

```
<[:label][.attribute.]field[>]
```

The syntax for defining screen attributes in Dialogue Manager is

```
<[&:label][.attribute.]&variable[>]
```

where:

.attribute.

Is one or more of the attributes. Note the dots (periods) before and after each attribute or entry in an attribute list.

field

Names the field to which the attributes apply.

&variable

Names the variable field to which the attributes apply.

Note: Labels and their use are discussed in *Using Labeled Fields* on page 252.

The following chart shows you how to use these attributes in conjunction with prefixes (D. and T.), where X is the name of a field or variable:

.HT.X	Highlighted T.
.CT.&X	Unhighlighted T.

.N.X	Nodisplay entry, (for example, for passwords)
.H.&X	Highlighted entry
.C.X	Unhighlighted entry
.HD.X	Highlighted D.

The following usage considerations apply when using screen attributes:

- An attribute stays in effect until another attribute changes it.
- A list of attributes may be composed entirely of abbreviations in any order. If abbreviations only are used, you do not need the dot separator between attributes.
- The last mentioned option in a group of mutually exclusive attributes will be taken.
- A color or flash overrides a highlight, clear, or Nodisplay.
- If short names are used, the first four letters identify the attribute. Each name must be separated by a dot. Either abbreviations or short names can be used, but they cannot be mixed without a dot separator.
- Full names may be used as well. Each must be delimited by a dot.
- You can change screen attributes during the course of a terminal session by using labeled fields.

Note the following examples:

.AID.	Aqua inverted display field.
<.RED.FLASH.	Red flashing field.
<.RED.FLAS.	Red flashing field.
<.PIN.	Inverted pink field (color overrides).
<I.YELL.	Inverted yellow field.

Using Background Effects

If a field is absent, the attribute affects the protected portion of the screen; that is, the text. Both a beginning and ending dot as well as a space between the attribute and the text are needed. For example:

```
"<.RED. ENTER EMP_ID:"
```

This will print the words ENTER EMP_ID: in red. Note the space between .RED. and ENTER EMP_ID:. A right caret may also be inserted for clarity.

The line:

```
"<.INVE.RED.   <.CLEAR.EMP_ID"
```

will turn the background color to red. CLEAR changes the background for the input field EMP_ID back to black.

An attribute stays in effect until another attribute changes it on a physical screen. Therefore, if <.INVE.RED. is in the upper left corner, the entire screen will be in inverse red unless some other background attribute is provided later. In the example above, the <.CLEAR is used to limit the effect to one area.

Note: .CLEAR. and .HIGH. only work when they are used in conjunction with a field. They do not work alone or simply with text.

Using Labeled Fields

You can use labels to identify a specific field on the screen. They are necessary to perform the following functions:

- Dynamically change screen attributes during processing depending on the results of tests.
- Position the cursor on the screen, or read the position of the cursor on the screen, where there is no pre-existing field.

The syntax for a labeled field in MODIFY is

```
<:label.field
```

The syntax for a labeled field in Dialogue Manager is

```
<&:label.&variable
```

where:

```
<[&]:label.
```

Is a user-defined label. It starts with a colon (:) and may be up to 66 characters long including the colon. You may not use embedded blanks.

field

Is any field on the CRTFORM. It can be a field created specifically for appending a label.

&variable

Is any variable field on the CRTFORM. It can be a field created specifically for appending a label.

The following rules apply:

- A label cannot occur by itself. It must be used with a field.
- A label must be declared using a COMPUTE, -SET, or -DEFAULTS statement.
- Setting a label to $ returns its field to the default attribute.

Example: **Using a Labeled Field With MODIFY**

For example, in MODIFY:

```
COMPUTE
:ONE/A6='   ';
CRTFORM
"<:ONE.EMP_ID"
```

The label :ONE is set to a format of A6 and is the identifier of the field EMP_ID.

Example: **Using a Labeled Field With Dialogue Manager**

For example, in Dialogue Manager:

```
-SET &:ONE='  ';
-CRTFORM
-"<&:ONE.&CITY/10"
```

In this Dialogue Manager example, the label &:ONE is set to a format of A4 and is the identifier of the field &CITY.

Note: When you are dealing with many complex labels and attributes, we advise you to use the FOCUS Screen Painter which allows you to do everything without learning the detailed syntax (see *Using the FOCUS Screen Painter* on page 302).

Dynamically Changing Screen Attributes

The screen attributes in a FIDEL form can be changed during the course of the terminal session in which they are defined. This allows you to design easy-to-read and easy-to-use procedures. For instance, after an error occurs, you can turn a specific field into flashing red to alert the operator.

The mechanism for changing the attribute is to put a label before the field. Then, issue a COMPUTE in MODIFY, or a -SET in Dialogue Manager, to assign the label new attribute values. When the screen is next displayed, it takes on the characteristics of the provided attributes.

The following example shows how to use a COMPUTE in MODIFY to dynamically change an attribute value:

```
COMPUTE
 :ATTRIB/A12=IF CURR_SAL GT 50000 THEN 'FLASH' ELSE '$';
CRTFORM
 "AMOUNT <:ATTRIB.T.CURR_SAL>"
IF CURR_SAL GT 50000 GOTO TOP ELSE GOTO OTHER;
 .
 .
 .
```

This generates an attribute value for the label ATTRIB. If the CURR_SAL is greater than 50,000, the field will flash; otherwise, it observes the default setting.

The following example shows the use of a -SET statement to assign an attribute value in Dialogue Manager:

```
-SET &AMOUNT=0;
-SET &:ATTRIB='     ';
-TOP
-CRTFORM
-"AMOUNT: <&:ATTRIB.T.&AMOUNT>"
-SET &:ATTRIB=IF &AMOUNT GT 100 THEN 'FLASH' ELSE '$';
-IF &AMOUNT GT 100 GOTO TOP;
 .
 .
 .
```

This generates an attribute value for the label &:ATTRIB, changing &AMOUNT to flashing if the value is greater than 100. Be sure to use -SET to establish the label in the beginning of the procedure.

Note: When you use CRTFORMs in either MODIFY or Dialogue Manager, the labels you assign must precede the fields with which they are associated; labels cannot occur by themselves. Use COMPUTE statements to dynamically change screen text attributes, setting the label equal to the COMPUTE (see previous example).

You can convert a T. field to a D. field dynamically; however, you cannot convert a D. field to a T. field. The method for changing turnaround fields to display fields is the same as that for changing screen attributes dynamically.

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
2.  "SALARY UPDATE"
2.  " "
3.  "EMPLOYEE ID #: <.INVE.EMP_ID LAST NAME: <0X
    <.CLEAR.D.LAST_NAME"
4.  MATCH EMP_ID
       ON NOMATCH REJECT
5.     ON MATCH CRTFORM LINE 10
6.     ENTER SALARY"
       " "
       "SALARY: <:HERE.T.CURR_SAL>"
7.  COMPUTE
       :HERE/A12=IF CURR_SAL GT 100000 THEN 'D' ELSE 'T';
       IF CURR_SAL GT 100000 GOTO TOP;
       ON MATCH UPDATE CURR_SAL
    DATA
    END
```

This procedure constructs a form to update salaries. It processes as follows:

1. CRTFORM generates the screen form and invokes FIDEL.
2. Provide text for the CRTFORM; empty quotation marks indicate a blank line on the form.
3. The next two lines contain the following screen elements:

   `EMPLOYEE ID #:`

   Is text describing the conditional data field EMP_ID.

   `.INVE.`

   Is a screen attribute that displays the field EMP_ID in reverse video.

   `LAST NAME:`

   Is text describing the field LAST_NAME.

   `.CLEAR.`

   Is a screen attribute that clears the .INVE. attribute, returning the D. (display-only) field LAST_NAME to the default display.
4. The request continues with MODIFY MATCH logic.
5. If EMP_ID matches, another CRTFORM is generated on line 10 of the same screen.
6. The next three lines contain the following screen elements:

   `ENTER SALARY:`

   Is text describing the CURR_SAL field.

   `" "`

   Generates a blank line.

`:HERE`

Is a label identifying the CURR_SAL field.

7. This COMPUTE evaluates the field CURR_SAL and defines it as a turnaround (T.) field or a display (D.) field, depending on the value of CURR_SAL. If the salary is greater than 100,000, the field is a display field (and cannot be updated); if the salary is less than 100,000, the field is a turnaround field (and can be updated).

The resulting CRTFORM is as follows:

```
SALARY UPDATE

EMPLOYEE ID #:     LAST NAME:




ENTER SALARY

SALARY:
```

Specifying Cursor Position

To specify cursor position, simply choose the field where you want the cursor positioned. You may specify the field by its field name or by its label. You can set the cursor at a specific place on the screen by computing or setting the value of the field CURSOR (in MODIFY) or &CURSOR (in Dialogue Manager).

The syntax for the field which controls the cursor position in MODIFY is

```
COMPUTE
CURSOR/A66= expression;
```

where:

`CURSOR/A66`

Is a 66-character alphanumeric field.

`expression`

Is terminated with a semicolon and can be anything, including the full field name, its full alias, or a unique truncation of either, or the label itself. This determines the position of the cursor.

For example:

```
COMPUTE
CURSOR/A66=IF TESTNAME GT 100 THEN 'EMP_ID'
ELSE 'LAST_NAME';
```

The position of the cursor will be on the field EMP_ID if the value of test name is greater than 100, or it will be on the field LAST_NAME if test name is less than or equal to 100.

You may also position the cursor using a field label. For example:

```
COMPUTE
CURSOR/A66=IF TESTNAME GT 100 THEN ':ONE'
ELSE ':TWO';
```

Note: If the field name is not unique, FIDEL uses the first occurrence of the field name (going from left to right across each line and then down to the next line) to set or test the cursor position.

In MODIFY, the variable CURSORINDEX can also be used to compute the position of the cursor at a particular record when there are multiple indexed records displayed in a single CRTFORM. This feature is commonly used for placing the cursor on invalid fields after VALIDATE statements. The syntax is

```
COMPUTE
CURSORINDEX/I5=expression;
```

where:

`CURSORINDEX/I5`

Is a five-digit integer field. Refers to the current value of the subscript being processed from the CRTFORM.

`expression`

May be any expression, but in most applications will be set equal to REPEATCOUNT.

Note: See *Case Logic, Groups, CURSORINDEX and VALIDATE* for a full example of the use of CURSORINDEX using Case Logic, multiple fields and the VALIDATE subcommand. Also, multiple record processing is discussed in full in *Multiple Record Processing* on page 169.

In Dialogue Manager, the syntax for positioning the cursor is

```
-SET &CURSOR=expression;
```

where:

`&CURSOR`

Is a variable specifically referring to the position of the cursor.

expression

Is terminated with a semicolon and can be any valid expression including the field name or label itself. It determines the position of the cursor.

The following example illustrates the positioning of the cursor on the screen in Dialogue Manager using labeled fields:

```
1.  -SET &:AAA = '     ';
    -SET &:BBB = '     ';
2.  -PROMPT &YR.PLEASE ENTER YEAR NEEDED.
3.  -SET &CURSOR = IF &YR GT 1984 THEN ':AAA' ELSE ':BBB';
    -*
4.  -CRTFORM
    -"MONTHLY REPORT FOR THE CITY <&:AAA.&CITY/10"
    -"YEARLY REPORT FOR THE AREA <&:BBB.&AREA/1"
        .
        .
        .
```

This processes as follows:

1. Two -SET statements declare the labels, which are themselves variables.
2. The -PROMPT statement prompts the operator for a value for &YR.
3. The -SET statement sets an IF test as the value for the variable &CURSOR. If the value of &YR is greater than 1984, the position of the cursor is set to the label :AAA; otherwise, it is set to the label :BBB.
4. If the operator supplies the value 85 for &YR, the visual form generated is as follows, and the cursor is positioned at the variable &CITY:

```
MONTHLY REPORT FOR THE CITY
YEARLY REPORT FOR THE AREA
```

The remainder of the FOCEXEC might then branch to a TABLE request for a monthly report for that city. Had the year been earlier than 84, the cursor would have been positioned on AREA. The branch might then be to a TABLE request for a yearly report for that area.

Caution: In Dialogue Manager, be sure to set &CURSOR to the label name without the & (ampersand). Use :AAA, not &:AAA.

Determining Current Cursor Position for Branching Purposes

Rather than having the operator type a response, you can create a menu on which you list options. To select an option, the operator moves the cursor to the correct line on the screen and presses the Enter key. FOCUS senses the cursor position and takes action based upon it (such as branching to a particular case or field).

To do this, you must specify a 66 character field that contains the current cursor position, CURSORAT. You may identify a field on the screen by a label or by its field name.

The syntax that defines the field used to read the cursor position in MODIFY is

```
COMPUTE
CURSORAT/A66=;
```

where:

`CURSORAT/A66`

Is the field whose value is determined by the field name, or label of the field, on which the cursor is positioned when the operator presses Enter.

In Dialogue Manager, the syntax is

```
-SET &CURSORAT='      ';
```

where:

`&CURSORAT`

Is a variable whose value is determined by the field name, or label of the field, on which the cursor is positioned when the operator presses Enter.

If the actual cursor position is not on any field, the value of CURSORAT is the nearest preceding field. If there are no preceding fields, the value of CURSORAT is the TOP of the CRTFORM. That is, the value is at the very beginning of the CRTFORM.

In the following example, field XYZ is a computed field for the purpose of creating a labeled field wherever necessary on the CRTFORM:

```
    MODIFY FILE EMPLOYEE
1.  COMPUTE
    CURSORAT/A66=;
2.  :ADD/A1=;
    :UPP/A1=;
3.  XYZ/A1=;
4.  CRTFORM
       "POSITION CURSOR NEXT TO OPTION DESIRED"
       "THEN PRESS ENTER"
       " "
       "<:ADD.XYZ   ADD RECORDS"
       "<:UPP.XYZ   UPDATE RECORDS"
5.  IF CURSORAT EQ ':ADD' GOTO ADD ELSE
    IF CURSORAT EQ ':UPP' GOTO UPP ELSE GOTO TOP;

     CASE ADD
    CRTFORM LINE 1
       "THIS CRTFORM ADDS RECORDS"
       " "
       "EMPLOYEE ID #: <EMP_ID"
       "LAST NAME:     <LAST_NAME"
       "FIRST NAME:    <FIRST_NAME"
       "HIRE DATE:     <HIRE_DATE"
       "DEPARTMENT:    <DEPARTMENT"
    MATCH EMP_ID
       ON MATCH REJECT
       ON NOMATCH INCLUDE
    ENDCASE

    CASE UPP
    CRTFORM LINE 1
       "THIS CRTFORM UPDATES RECORDS"
       " "
       "EMPLOYEE ID #: <EMP_ID"
       "DEPARTMENT:    <DEPARTMENT"
       "JOB CODE:      <CURR_JOBCODE"
       "SALARY:        <CURR_SAL"
    MATCH EMP_ID
       ON NOMATCH REJECT
       ON MATCH UPDATE DEPARTMENT CURR_JOBCODE CURR_SAL
    ENDCASE
    DATA
    END
```

This example processes as follows:

1. The COMPUTE establishes the field CURSORAT.
2. The second and third COMPUTEs declare the labels :ADD and :UPP.
3. The third COMPUTE establishes a field XYZ for the purpose of using labels.

4. CRTFORM generates the following visual form beginning on the first line of the screen:

```
POSITION CURSOR NEXT TO OPTION DESIRED
THEN PRESS ENTER

ADD RECORDS
UPDATE RECORDS
```

5. An IF phrase tests the value of the field CURSORAT. If the operator places the cursor next to ADD RECORDS, the value of CURSORAT is :ADD and a branch to Case ADD will be performed. If the operator places the cursor next to UPDATE RECORDS, the value of CURSORAT is :UPP and Case UPP will be performed.

Annotated Example: MODIFY

The following example illustrates the syntax for a MODIFY CRTFORM using dynamically changing attributes:

```
MODIFY FILE EMPLOYEE
1. CRTFORM
2. "EMPLOYEE UPDATE"
3. "</1"
4. "EMPLOYEE ID #: <.INVE.EMP_ID"
GOTO UPDATE
CASE UPDATE
5. MATCH EMP_ID
ON NOMATCH REJECT
6. ON MATCH CRTFORM LINE 1
" "
7. "LAST NAME: <.INVE.T.LAST_NAME"
"DEPARTMENT: <.CLEAR.T.DEPARTMENT>"
"SALARY: <:ATTRIB.T.CURR_SAL>"
8. ON MATCH COMPUTE
:ATTRIB/A12 = IF CURR_SAL GT 50000 THEN 'FLASH.INVE';
MSG/A60 = IF CURR_SAL GT 50000 THEN 'PLEASE REENTER' ELSE ' ';
ON MATCH TYPE "<MSG"
ON MATCH IF CURR_SAL GT 50000 GOTO UPDATE;
ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
ENDCASE
DATA
END
```

This procedure sets up a form to update the department and current salary. It processes as follows:

1. CRTFORM generates the visual form and invokes FIDEL.
2. This line contains a screen element set between double quotations marks. It is a line of descriptive text.
3. This line contains another screen element, a spot marker that skips one line.

4. These lines contain four screen elements—'EMPLOYEE ID #:' is text describing the field; the field EMP_ID is described as a conditional data entry field. The contents will be displayed in reverse video because .INVE. is a screen attribute defining the field.

   The visual form generated is as follows:

   ```
   EMPLOYEE UPDATE
   EMPLOYEE ID #: (reverse video)
   ```

   Enter Employee ID # 818692173.

5. The request continues with MODIFY MATCH logic.
6. If the EMP_ID matches, another CRTFORM is generated. It is placed on LINE 1 and thus replaces the previous CRTFORM on the screen.
7. The CRTFORM defines three turnaround fields:

   The LAST_NAME field. The .INVE. attribute displays the field in reverse video.

   The DEPARTMENT field. The .CLEAR. attribute displays the field in regular video.

   The CURR_SAL field. The appearance of the field value depends on the value of the :ATTRIB field. When the CURR_SAL value first appears, the :ATTRIB field is empty and the value appears in regular video. If you enter a CURR_SAL value greater than 50,000, the :ATTRIB field receives the attribute FLASH.INVE, displaying the CURR_SAL value in flashing inverse (or reverse) video. The CRTFORM appears as follows:

   ```
   LAST NAME:   CROSS
   DEPARTMENT:  MIS
   SALARY:      27062.00
   ```

8. If the CURR_SAL field value is greater than 50,000 when you press Enter, the COMPUTE statement assigns the :ATTRIB label the FLASH.INVE attribute.
9. If the CURR_SAL field value is greater than 50,000 when you press Enter, the IF statement branches back to the CASE UPDATE statement. This displays the second CRTFORM with the CURR_SAL value in reverse video.

   Note: If you are using a terminal emulator you may not be able to view the attribute FLASH.INVE.

Annotated Example: Dialogue Manager

The following sample -CRTFORM illustrates the syntax for dynamic control of attributes in Dialogue Manager:

```
1.  -PROMPT &CITY.FOR WHICH CITY DO YOU WANT A REPORT?.
2.  -SET &:ATTRIB = IF &CITY EQ STAMFORD THEN 'INVE' ELSE 'CLEAR';
    -*
3.  -CRTFORM
4.  -"MONTHLY SALES REPORT"
5.  -"Date: <D.&DATE      Time: <D.&TOD"
6.  -"Beginning Code is:  <&:ATTRIB.&BEGCODE/3"
    -"Ending Code is:   <&:ATTRIB.&ENDCODE/3"
    -"Regional Supervisor is:  <&:ATTRIB.&REGIONMGR/15"
    TABLE FILE SALES
    HEADING CENTER
    "MONTHLY REPORT FOR &CITY"
    "PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
    " "
    SUM UNIT_SOLD AND RETURNS AND COMPUTE
    RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
    BY PROD_CODE
    IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
    IF CITY EQ &CITY
    FOOTING CENTER
    "REGION MANAGER: &REGIONMGR"
    "CALCULATED AS OF &DATE"
7.  END
```

The example processes as follows:

1. The -PROMPT prompts the operator for a value for &CITY.
2. The -SET statement sets the label :ATTRIB to INVE if the city is Stamford, causing each field labeled :ATTRIB in the remainder of the -CRTFORM to be displayed in reverse video.
3. -CRTFORM generates the visual form and invokes FIDEL.
4. The first line of the screen form contains text.
5. The second line contains the current date and time as display fields. Since they are in protected areas of the screen, they cannot be altered.
6. Each of the next three lines contains descriptive text and one field. Each field has a label which displays the field in reverse video if the city is Stamford.

   The screen displays the following -CRTFORM:

```
MONTHLY SALES REPORT
Date: 01/08/97    Time: 10:50:16
Beginning Code is:
Ending Code is:
Regional Supervisor is:
```

7. After the operator presses Enter, the values entered in the screen form are sent to the variables. The TABLE request executes when END is encountered.

Using FIDEL in MODIFY

The following standard MODIFY functions and concepts work with FIDEL in the building of CRTFORMs (for additional information on these functions):

- Conditional and non-conditional field specification (see *Conditional and Non-Conditional Fields* on page 264).
- The FIXFORM statement which can be used before the first CRTFORM. This enables you to mix data sources (see *Using FIXFORM and FIDEL in a Single MODIFY* on page 268).
- Automatic application generation, which enables you to use several simple statements to generate automatic data management procedures and CRTFORMs (see *Generating Automatic CRTFORMs* on page 270).
- Multiple CRTFORMs for different processing options. The additional FIDEL facility of the LINE option helps you organize the use of multiple CRTFORMs (see *Using Multiple CRTFORMs: LINE* on page 274).
- Case Logic, which enables you to divide the processing into logical subdivisions for particular sets of circumstances (see *Case Logic* on page 145, and *CRTFORMs and Case Logic* on page 279).
- Groups of fields (see *Specifying Groups of Fields* on page 281).
- VALIDATES and various error handling formats (see *Handling Errors* on page 289).
- Log files that preserve a record of all data that is entered onto the screen (see *Logging Transactions* on page 293).

MODIFY also has additional screen control options such as clearing the screen, setting the height and width parameters, and changing the default size of the TYPE message area in order to enlarge the CRTFORM (see *Additional Screen Control Options* on page 293).

Conditional and Non-Conditional Fields

When you run a MODIFY request, FOCUS keeps track of which transaction fields are active or inactive during execution. In order to add, update, and delete segment instances, the fields must be active (see *Active and Inactive Fields* on page 204, for a full discussion of active and inactive fields).

You can define data entry and turnaround fields as either conditional or non-conditional. A conditional field is conditionally active. That is, it becomes active only if there is incoming data present for the field. Otherwise, it remains inactive. A non-conditional field is always active whether there is incoming data present or not.

When you are performing update operations, there are several important points to keep in mind when you choose whether to specify a field as conditional or non-conditional:

- If data is entered or changed, the data source value is always updated and the field always becomes active. This is true whether the field is conditional or non-conditional.
- If data is not entered or changed, what happens to the data source value is dependent on whether the field is conditional or non-conditional as well as program logic. The following table outlines this.

Type of Field	Active/Inactive	Data Source Value
Conditional Entry	Inactive	Remains. Display value ignored.
Conditional Turnaround	Inactive	Remains. Display value ignored.
Non-Conditional Entry	Active	Displayed value replaces data source value (blank or 0).
Non-Conditional Turnaround	Active	Displayed value replaces data source value (same value).

- If a field is active, the displayed value always becomes the new data source value. In turnaround fields, this is by definition the same value.
- If a field is inactive, the displayed value is always ignored.
- If you compute a data source field and then display it on the CRTFORM with a D. or a T., the field must still be active to get the computed value displayed on the screen. Otherwise, you get a blank or 0.
- When you use a VALIDATE for a field, the field must be active. Otherwise you do not get the accurate data source value validated; instead, you get a blank or 0.

  Note: You can make a field active or inactive by using the ACTIVATE or DEACTIVATE statement respectively.

Example: Conditional and Non-Conditional Display and Turnaround Fields

The following example illustrates the display and turnaround field features as well as the use of a non-conditional turnaround field (both carets). The first CRTFORM asks for a key field value, in this case EMP_ID. If a matching record is obtained, then some data source values are displayed and others are shown for turnaround update.

Note how the non-conditional turnaround field functions in the following example. Whether the displayed value is changed or not, the value in the data source is active. The VALIDATE uses the display value, whether it was changed or not.

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
       "ENTER EMPLOYEE ID#: <EMP_ID"
       "PRESS ENTER BEFORE CONTINUING"
       "-------------------------------------------------"
    MATCH EMP_ID
       ON NOMATCH TYPE
          "EMPLOYEE ID NOT IN DATABASE. PLEASE REENTER."
       ON NOMATCH REJECT
2.     ON MATCH CRTFORM LINE 4
          " "
          "EMPLOYEE ID #:  <D.EMP_ID"
          "LAST NAME:  <D.LAST_NAME"
          "HIRE DATE:  <D.HIRE_DATE"
          "SALARY:  <T.CURR_SAL>"
          "DEPARTMENT:  <T.DEPARTMENT>"
3.     ON MATCH VALIDATE
          SALTEST = IF CURR_SAL GT 0 THEN 1 ELSE 0;
          ON INVALID TYPE
          "SALARY MUST BE GREATER THAN 0"
          "CORRECT SALARY AND PRESS ENTER TWICE"
       ON MATCH UPDATE CURR_SAL DEPARTMENT
    DATA
    END
```

The example processes as follows:

1. When the procedure executes, the top part of the CRTFORM appears as follows:

```
ENTER EMPLOYEE ID #:
PRESS ENTER BEFORE CONTINUING
-----------------------------------------------------
```

   If the employee ID entered does not match an ID in the data source, the transaction is rejected and a TYPE statement appears at the bottom of the screen.

   Assume the operator enters the following employee ID:

```
ENTER EMPLOYEE ID #: 818692173
PRESS ENTER BEFORE CONTINUING
-----------------------------------------------------
```

2. If the ID entered matches an ID in the data source, FOCUS successfully retrieves a record. The ON MATCH CRTFORM causes a second CRTFORM to be displayed on line 4. This CRTFORM contains both display and turnaround fields. The data source values of the fields appear on the second CRTFORM, and the cursor is positioned at the start of the CURR_SAL field which is the first unprotected field. Note that both CURR_SAL and DEPARTMENT are automatically highlighted for turnaround:

```
ENTER EMPLOYEE ID #: 818692173
PRESS ENTER BEFORE CONTINUING
-------------------------------------------------------

EMPLOYEE ID #:   818692173
LAST NAME:       CROSS
HIRE DATE:       811102
SALARY:          27062.00
DEPARTMENT:      MIS
```

Assume the operator updates DEPARTMENT, does not change CURR_SAL, and transmits the CRTFORM:

```
ENTER EMPLOYEE ID #: 818692173
PRESS ENTER BEFORE CONTINUING
-------------------------------------------------------

EMPLOYEE ID #:   818692173
LAST NAME:       CROSS
HIRE DATE        811102
SALARY:          27062.00
DEPARTMENT:      ois
```

3. When the operator presses Enter, the transaction is processed. If the value of CURR_SAL is greater than 0, the VALIDATE will evaluate as 1 (true) and processing continues. Although a value was not entered for CURR_SAL, the field is active because it is specified as a non-conditional field. Thus, the VALIDATE reads the current value in the T. field (27062.00), and validates the field. The transaction is then processed.

If you specify the turnaround field as conditional (only the left caret), the field is inactive if no data is entered. Assume the same transaction as above. The operator updates the DEPARTMENT and does not enter new data for the CURR_SAL field. The VALIDATE does not read the T. value because the field is inactive and instead reads a 0. The field is invalidated and the following error message occurs:

```
ENTER EMPLOYEE ID #: 818692173
PRESS ENTER BEFORE CONTINUING
--------------------------------

EMPLOYEE ID #:   818692173
LAST NAME:       CROSS
HIRE DATE:       811102
SALARY:          27062.00
DEPARTMENT:      ois

(FOC421)TRANS 1 REJECTED INVALID SALTEST
INVALID SALARY
SALARY MUST BE GREATER THAN 0
```

Using FIXFORM and FIDEL in a Single MODIFY

A MODIFY procedure can mix data sources from CRTFORMs and FIXFORMs.

The rules are:

- You can have only one FIXFORM statement and you must specify the name of the transaction data source. For example:

  ```
  FIXFORM ON filename
  ```

- The FIXFORM statement must precede the CRTFORM statement.
- START and STOP do not apply (see *Reading Selected Portions of Transaction Data Sources: The START and STOP Statements* on page 73).

This feature is useful in situations where a known set of records is wanted and the keys for these records reside on an external fixed format data source. (The data source may have been produced by a prior TABLE and SAVE or HOLD command.) The procedure first reads a key, fetches the matching record, and displays it on the CRTFORM specified.

This is also convenient when the FIXFORM is included in a START case.

In the following example, FIXFORM is used with FIDEL. To run this example on your machine, you must first create a sequential data source with data. To do so, run this TABLE request:

```
TABLE FILE EMPLOYEE
PRINT EMP_ID PAY_DATE
IF PAY_DATE GE 820730
ON TABLE SAVE AS PAYTRANS
END
```

This creates the transaction data source PAYTRANS. Then run the following MODIFY request:

```
    MODIFY FILE EMPLOYEE
1.  FIXFORM ON PAYTRANS EMP_ID/9 PAY_DATE/6
2.  MATCH EMP_ID
      ON NOMATCH REJECT
      ON MATCH CONTINUE
    MATCH PAY_DATE
3.    ON MATCH/NOMATCH CRTFORM
         "EMPLOYEE ID #:<D.EMP_ID"
         "PAY DATE:<D.PAY_DATE"
         "MONTHLY GROSS:<T.GROSS>"
      ON NOMATCH INCLUDE
      ON MATCH UPDATE GROSS
    DATA
    END
```

The example processes as follows:

1. First the data is read in from the sequential data source PAYTRANS.
2. The EMP_ID from PAYTRANS is matched against EMP_IDs in the EMPLOYEE data source. If the EMP_IDs match, PAY_DATE is matched.
3. The CRTFORM shows display values for EMP_ID and PAY_DATE. If there is a match on PAY_DATE, GROSS is displayed as a turnaround field and the operator can update it. If there is no match on PAY_DATE, both PAY_DATE and GROSS are included:

```
EMPLOYEE ID #:   071382660
PAY_DATE:        820831
MONTHLY GROSS:         916.67
```

The procedure ends when there are no more transactions to read on the external data source. It can also be terminated by the operator by pressing the *PF1* or *PF3* key.

Generating Automatic CRTFORMs

You can use several simple but powerful statements with the FOCUS MODIFY facility to allow immediate generation of data management requests. You do not need to learn the complete FOCUS MODIFY language. Without using field names, you can write general-purpose requests and customize them for more detailed situations.

The statements can be used with multi-segment data sources as well as simple data sources. They can also be used from the Screen Painter (see *Generating CRTFORMs Automatically* on page 312). These statements automatically specify conditional fields. They include:

`CRTFORM * [SEG n]`	Design screen for all real data fields in segment *n*, where *n* is either the segment name or number.
`CRTFORM * KEYS [SEG n]`	Design screen for all key fields in segment *n*.
`CRTFORM * NONKEYS [SEG n]`	Design screen for all non-key fields in segment *n*.
`CRTFORM T.* [SEG n]`	Design screen using T.fields in segment *n*
`CRTFORM D.* [SEG n]`	Design screen using D.fields in segment *n*.

Note: The use of CRTFORM * on a COMBINE data source name is illogical and may produce unpredictable results.

Note that you can optionally specify the segment name or number for each of the CRTFORMs. To obtain the segment names and numbers, enter the following command where *file* is the name of the data source:

```
CHECK FILE file PICTURE
```

The names and numbers appear on the top of each segment in the diagram. You may also list segment names and numbers by entering the command:

```
? FDT filename
```

See the *Describing Data* manual and the *Developing Applications* manual for more information on the CHECK FILE command and ? FDT query.

If you are modifying all of the segments in the data source (except for unique segments), you can write the request without using Case Logic. The following example adds and maintains data for the EMPLOYEE data source. The segments are as follows:

- Segment 1 contains basic employee data (names, jobs, salaries, and so on).
- Segment 3 contains employee salary histories.
- Segment 7 stores employees' home addresses and information on their bank accounts.
- Segment 8 stores employee monthly pay.
- Segment 9 stores monthly deductions.

(Segment 2 is a unique segment. Segments 4, 5, and 6 are cross-referenced segments that are not part of the EMPLOYEE data source.)

The request is:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "THIS PROCEDURE ADDS NEW RECORDS AND UPDATES EXISTING RECORDS </1"
   "INSTRUCTIONS"
   "1. ENTER DATA FOR EACH FIELD"
   "2. USE TAB KEY TO MOVE CURSOR"
   "3. PRESS ENTER WHEN FINISHED"
   "4. WHEN YOU FINISH ALL RECORDS, PRESS PF1 </1"
CRTFORM * KEYS
MATCH * KEYS SEG 01
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 01
   ON MATCH UPDATE * SEG 01
   ON NOMATCH INCLUDE
MATCH * KEYS SEG 03
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 03
   ON MATCH UPDATE * SEG 03
   ON NOMATCH INCLUDE
MATCH * KEYS SEG 07
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 07
   ON MATCH UPDATE * SEG 07
   ON NOMATCH INCLUDE
MATCH * KEYS SEG 08
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 08
   ON MATCH UPDATE * SEG 08
   ON NOMATCH INCLUDE
MATCH * KEYS SEG 09
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 09
   ON MATCH UPDATE * SEG 09
   ON NOMATCH INCLUDE
DATA
END
```

When the procedure executes, the screen appears as follows:

```
THIS PROCEDURE ADDS NEW RECORDS AND UPDATES EXISTING RECORDS

INSTRUCTIONS
1. ENTER DATA FOR EACH FIELD
2. USE TAB KEY TO MOVE CURSOR
3. PRESS ENTER WHEN FINISHED
4. WHEN YOU FINISH ALL RECORDS, PRESS PF1

EMP_ID:      :
DAT_INC:     :
TYPE:    :
PAY_DATE:     :
DED_CODE:    :

LAST_NAME:        :                    FIRST_NAME:        :
HIRE_DATE:     :                       DEPARTMENT:        :
CURR_SAL:       :                      CURR_JOBCODE:    :
ED_HRS:     :

PCT_INC:     :                         SALARY:        :
JOBCODE:   :

ADDRESS_LN1:          :
ADDRESS_LN2:          :
ADDRESS_LN3:          :

ACCTNUMBER:      :

GROSS:       :
```

Notice that the fields are divided into five groups. The first group consists of all the key fields in the data source. Each subsequent group consists of all non-key fields in a particular segment. Fill in each group from top to bottom and press Enter before filling in the next group. When you do this, the request uses the values to match on the segments specified later in the request.

The first CRTFORM statement generates the first group of fields, which are all the key fields in the data source:

```
CRTFORM * KEYS
```

The MATCH statements in the request modify each of the segments in the data source. Each statement contains a CRTFORM phrase that prompts for all non-key fields in the segment:

```
CRTFORM T.* NONKEYS SEG xx
```

Note that the CRTFORM phrase displays the fields as turnaround fields. After you fill in the fields in the group and press Enter, FOCUS uses the field values to update the segment.

You can add the following enhancements to the request:

- The LINE option on each CRTFORM statement.
- Explanatory text after each CRTFORM statement.
- A separate CRTFORM containing explanatory text at the beginning of the request.

If you want to modify some but not all segments in the data source, use Case Logic to write the request. Place each MATCH statement in a separate case. For example, this request modifies data in Segments 1, 3, and 7:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "THIS PROCEDURE MAINTAINS EMPLOYEE"
   "JOB DATA, SALARY HISTORIES, AND ADDRESSES"
   " "
CRTFORM * KEYS
   "FILL IN EMP_ID, DAT_INC, AND TYPE FIELDS"
   "THEN PRESS ENTER"
GOTO EMPLOYEE

CASE EMPLOYEE
MATCH * KEYS SEG 01
   ON NOMATCH REJECT
   ON MATCH CRTFORM T.* NONKEYS SEG 01 LINE 10
   ON MATCH UPDATE * SEG 01
   ON MATCH GOTO MONTHPAY
ENDCASE

CASE MONTHPAY
MATCH * KEYS SEG 03
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 03 LINE 10
   ON MATCH UPDATE * SEG 03
   ON MATCH GOTO DEDUCT
   ON NOMATCH INCLUDE
   ON NOMATCH GOTO DEDUCT
ENDCASE

CASE DEDUCT
MATCH * KEYS SEG 07
   ON MATCH/NOMATCH CRTFORM T.* NONKEYS SEG 07 LINE 10
   ON MATCH UPDATE * SEG 07
   ON NOMATCH INCLUDE
ENDCASE
DATA
END
```

Using Multiple CRTFORMs: LINE

You can choose which screen line the CRTFORM will begin on by using the LINE option. By default, the first CRTFORM begins on line 1. The next CRTFORM in the procedure begins on the line following the end of the previous CRTFORM. For example, if one screen uses 12 lines, the next CRTFORM automatically begins on the 13th line.

In the following example, there are two logical forms: EMPLOYEE UPDATE and FUND TRANSFER INFORMATION UPDATE. It illustrates the placement of CRTFORMs when the default is in effect (that is, the LINE option is not used):

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
       "EMPLOYEE UPDATE"
       " "
       "-----------------------------------------------"
       "EMPLOYEE ID #: <EMP_ID    LAST_NAME: <LAST_NAME"
       "
       "DEPARTMENT:    <DEPARTMENT <28 SALARY: <CURR_SAL"
       " "
       "BANK: <BANK_NAME"
       " "
       "FILL IN THE ABOVE FORM AND PRESS ENTER"
       "-----------------------------------------------"
    MATCH EMP_ID
       ON NOMATCH REJECT
       ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
       ON MATCH CONTINUE TO BANK_NAME
         ON NOMATCH INCLUDE
2.       ON MATCH/NOMATCH CRTFORM
       "</1"
       "FUND TRANSFER INFORMATION UPDATE"
       " "
       "-----------------------------------------------"
       "BANK: <D.BN   ACCT #: <T.BA"
       " "
       "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
       "-----------------------------------------------"
           ON MATCH UPDATE BA BC EDATE
DATA
END
```

This produces the following screen when the request is executed:

```
EMPLOYEE UPDATE

---------------------------------------------------------
EMPLOYEE ID #:           LAST_NAME:

DEPARTMENT:              SALARY:

BANK:

FILL IN THE ABOVE FORM AND PRESS ENTER
---------------------------------------------------------

FUND TRANSFER INFORMATION UPDATE

---------------------------------------------------------
BANK:                    ACCT #:

BANK CODE:               START DATE:
---------------------------------------------------------
```

Note that when the default is in effect, each logical form is displayed one after the other on the screen, the instant the MODIFY procedure is executed. That is, all the distinct CRTFORMs are concatenated into one visual form.

The LINE option enables you to control both the placement of a CRTFORM on the screen and the timing with which it appears on the screen. Using LINE gives you the following options:

- You can have one logical form replace another after each transaction by having subsequent CRTFORMs begin on the same line.
- You can build mixed screens by saving lines from a previous CRTFORM on the screen while executing a subsequent CRTFORM. In other words, the first CRTFORM is displayed, the operator transmits the data, and the next CRTFORM is displayed while the previous one remains on the screen.

The syntax is

```
CRTFORM [LINE nn]
```

where:

nn

Is the starting line number for the CRTFORM.

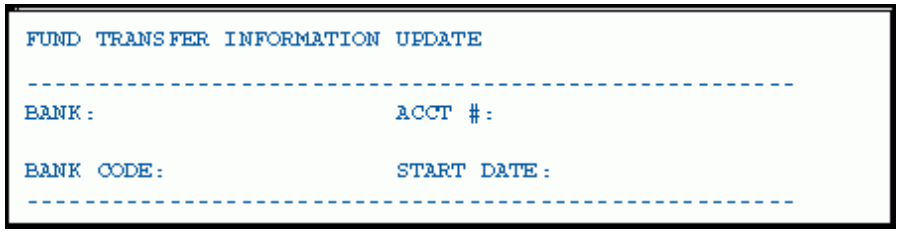
To completely replace one screen with the next, both CRTFORMs must start on the same line. Note the following change in the previous example:

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
       "EMPLOYEE UPDATE"
       " "
       "-------------------------------------------------"
       "EMPLOYEE ID #:  <EMP_ID LAST_NAME: <LAST_NAME"
       " "
       "DEPARTMENT:     <DEPARTMENT <30 SALARY: <CURR_SAL"
       " "
       "BANK:           <BANK_NAME"
       " "
       "FILL IN THE ABOVE FORM AND PRESS ENTER"
       "-------------------------------------------------"
    MATCH EMP_ID
      ON NOMATCH REJECT
      ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
      ON MATCH CONTINUE TO BANK_NAME
      ON NOMATCH INCLUDE

2.    ON MATCH/NOMATCH CRTFORM LINE 1
      "</1"
      "FUND TRANSFER INFORMATION UPDATE"
      " "
      "-------------------------------------------------"
      "BANK: <D.BN ACCT #: <T.BA"
      " "
      "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
      "-------------------------------------------------"
      ON MATCH UPDATE BA BC EDATE
    DATA
    END
```

1. When the MODIFY procedure is executed, the following screen is displayed:

```
EMPLOYEE UPDATE

-------------------------------------------------------

EMPLOYEE ID #:            LAST_NAME:

DEPARTMENT:               SALARY:

BANK:

FILL IN THE ABOVE FORM AND PRESS ENTER
-------------------------------------------------------
```

2. After the operator enters and transmits the data, the next CRTFORM replaces the previous one on the screen:



Generally, it is a good practice to put LINE 1 on all CRTFORMs that start a new case (see *CRTFORMs and Case Logic* on page 279) unless a specific screen pattern is wanted.

A combination of two or more individual CRTFORMs can occupy specific lines on one screen. To obtain a mixed screen, place the desired starting line number with the CRTFORM statement. For instance:

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
      "EMPLOYEE UPDATE"
      " "
      "-----------------------------------------------"
      "EMPLOYEE ID #:    <EMP_ID LAST_NAME: <LAST_NAME"
      " "
      "DEPARTMENT:       <DEPARTMENT <30 SALARY: <CURR_SAL"
      " "
      "BANK:             <BANK_NAME"
      " "
      "FILL IN THE ABOVE FORM AND PRESS ENTER"

      "-----------------------------------------------"


    MATCH EMP_ID
      ON NOMATCH REJECT
      ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
      ON MATCH CONTINUE TO BANK_NAME
      ON NOMATCH INCLUDE
2.    ON MATCH/NOMATCH CRTFORM LINE 12
      "</1"
      "FUND TRANSFER INFORMATION UPDATE"
      " "
      "-----------------------------------------------"
      "BANK: <D.BN  ACCT #: <T.BA"
      " "
      "BANK CODE: <T.BC <30 START DATE: <T.EDATE"
      "-----------------------------------------------"
      ON MATCH UPDATE BA BC EDATE
DATA
END
```

Processing occurs as follows:

1. Like the preceding examples, this produces the first screen. Assume the operator enters and transmits the following data:

```
EMPLOYEE UPDATE

------------------------------------------------------
EMPLOYEE ID #:   117593129          LAST_NAME:   JONES

DEPARTMENT:   MIS                   SALARY:   18480

BANK:   STATE

FILL IN THE ABOVE FORM AND PRESS ENTER
------------------------------------------------------
```

2. The first CRTFORM remains on the screen while the next CRTFORM is displayed on line 12:

```
EMPLOYEE UPDATE

------------------------------------------------------
EMPLOYEE ID #:   117593129           LAST_NAME:   JONES

DEPARTMENT:   MIS                    CURRENT SALARY:   18480

BANK:   STATE

FILL IN THE ABOVE FORM AND PRESS ENTER
------------------------------------------------------

FUND TRANSFER INFORMATION UPDATE
------------------------------------------------------
BANK: STATE                          ACCT #:40950036

BANK CODE: 510271                    START DATE:821101

------------------------------------------------------
```

You can save certain lines from the preceding CRTFORM while you discard others. In the previous example, if you begin the second CRTFORM on line 6, the EMP_ID and the LAST_NAME remain and the next line is the beginning of the FUND TRANSFER INFORMATION AND UPDATE.

Assume the operator enters and transmits data on the first CRTFORM. Part of the first logical form disappears and the second form is displayed. Thus, a new visual form is created:

```
EMPLOYEE UPDATE

-----------------------------------------------------
EMPLOYEE ID #:   117593129        LAST_NAME:   JONES

FUND TRANSFER INFORMATION AND UPDATE

-----------------------------------------------------
BANK:  STATE                      ACCT #:   40950036

BANK CODE:  510271                START DATE:   821101
-----------------------------------------------------
```

You can create mixed screens using the LINE option, in a variety of ways, depending on the need of the application.

CRTFORMs and Case Logic

Case Logic, described in *Case Logic* on page 145, enables you to perform separate complete MODIFY processes in one procedure. Each of these is a case, and the procedure contains directions about which case to execute under various circumstances.

When you use the Case Logic option of the MODIFY command, you can create a pattern of many CRTFORMs.

When there are multiple CRTFORMs in a single MODIFY request, use the LINE option to specify where each CRTFORM will be displayed. With Case Logic, generally, we recommend that you use LINE 1 to replace one screen with another.

The following example illustrates the use of Case Logic with the CRTFORM:

```
MODIFY FILE EMPLOYEE
COMPUTE
   PFKEY/A4= ;
CRTFORM
   "TO INPUT A NEW RECORD, PRESS PF4"
   "TO UPDATE AN EXISTING RECORD, PRESS PF5"
IF PFKEY EQ 'PF04' GOTO ADD ELSE
IF PFKEY EQ 'PF05' GOTO UPP ELSE GOTO TOP;

CASE ADD
CRTFORM LINE 1
   "EMPLOYEE ID #:    <EMP_ID"
   "LAST NAME:   <LAST_NAME FIRST NAME: <FIRST_NAME"
   "HIRE DATE:   <HIRE_DATE"
   "DEPARTMENT:       <DEPARTMENT"
MATCH EMP_ID
   ON MATCH REJECT
   ON NOMATCH INCLUDE
ENDCASE

CASE UPP
CRTFORM LINE 1
   "EMPLOYEE ID #:    <EMP_ID"
   "DEPARTMENT:       <DEPARTMENT"
   "JOB CODE:    <CURR_JOBCODE"
   "SALARY:      <CURR_SAL"
MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH UPDATE DEPARTMENT CURR_JOBCODE CURR_SAL
ENDCASE
DATA
END
```

The first CRTFORM appears as:

```
TO INPUT A NEW RECORD, PRESS PF4
TO UPDATE AN EXISTING RECORD, PRESS PF5
```

If the operator presses PF4, the following is displayed:

```
EMPLOYEE ID #:
LAST NAME:              FIRST NAME:
HIRE DATE:
DEPARTMENT:
```

If the operator presses PF5, the following is displayed:

```
EMPLOYEE ID #:
DEPARTMENT:
JOB CODE:
SALARY:
```

Note: At the end of a MODIFY procedure, control defaults to the TOP Case.

Specifying Groups of Fields

Groups of fields (that is, more than one occurrence of the same field) can be specified on the CRTFORM in two ways:

- Specifying a field more than once on a CRTFORM.
- Using REPEAT syntax.

You can use Case Logic to process groups of fields.

Specifying Groups of Fields for Input

A group of fields may repeat on the form. For example:

```
"EMPLOYEE ID   DEPARTMENT   SALARY"
"<EMP_ID       <DPT         <CURR_SAL"
"<EMP_ID       <DPT         <CURR_SAL"
"<EMP_ID       <DPT         <CURR_SAL"
```

This reads the same data as the FIXFORM statement:

```
FIXFORM 3(EMP_ID/C9 DPT/C10 CURR_SAL/C14)
```

The following example shows the use of repeating groups for a single employee ID:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "ENTER EMPLOYEE ID #: <EMP_ID"
   " "
   "ENTER PAY DATE AND GROSS PAY FOR ABOVE EMPLOYEE"
   " "
   "PAY DATE: <PAY_DATE      GROSS: <GROSS"
   "PAY DATE: <PAY_DATE      GROSS: <GROSS"
   "PAY DATE: <PAY_DATE      GROSS: <GROSS"
MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH CONTINUE
MATCH PAY_DATE
   ON MATCH REJECT
   ON NOMATCH INCLUDE
DATA
END
```

Note: A group of repeated data fields cannot be specified on a MATCH or NOMATCH CRTFORM. They must be presented on a primary CRTFORM (that is, one not generated as a result of a MATCH or NOMATCH command).

This procedure processes as follows:

```
ENTER EMPLOYEE ID #:  818692173

ENTER PAY DATE AND GROSS AMOUNT FOR ABOVE EMPLOYEE

PAY DATE: 850405      GROSS: 3000.00
PAY DATE: 850412      GROSS: 4000.00
PAY DATE: 850418      GROSS: 2500.00
```

When the operator presses *Enter*, the transaction processes. Processing continues until a line with no data is found or all lines are completed (whichever occurs first).

Using REPEAT to Display Multiple Records

You can display multiple segment instances on the screen by directing FIDEL to read and display the contents of a HOLD buffer. You can use a subscript value to identify a particular instance in the HOLD buffer with the following syntax

field(*n*)

where:

field

Is the name of a previously held field.

(*n*)

Is the integer subscript that identifies the number of the instance in the HOLD buffer containing the field to be displayed. *n* must be in integer format or the report group will be ignored.

The variable SCREENINDEX allows you to display and modify selected groups of records from the HOLD buffer.

Consider the following example, which uses the REPEAT statement to retrieve up to a set number (in this case, six) of multiple instances, saves them in the HOLD buffer, and then displays the instances on the CRTFORM:

```
MODIFY FILE EMPLOYEE
1.  CRTFORM
       "PAY HISTORY UPDATE"
       " "
       "ENTER EMPLOYEE ID#: <EMP_ID"
    MATCH EMP_ID
       ON NOMATCH REJECT
       ON MATCH GOTO COLLECT

    CASE COLLECT
2.  REPEAT 6 TIMES
2.     NEXT PAY_DATE
2.        ON NEXT HOLD PAY_DATE GROSS
3.        ON NONEXT GOTO DISPLAY

3.  ENDREPEAT
    GOTO DISPLAY
    ENDCASE
```

```
CASE DISPLAY
    IF HOLDCOUNT EQ 0 GOTO TOP;
4.  COMPUTE
       BUFFNUMBER/I5 = HOLDCOUNT;
5.  CRTFORM LINE 5
       "FILL IN GROSS AMOUNT FOR EACH PAY DATE"
       " "
       "PAY DATE          GROSS AMOUNT"
       "--------          ------------"
       "<D.PAY_DATE(1)    <T.GROSS(1)>"
       "<D.PAY_DATE(2)    <T.GROSS(2)>"
       "<D.PAY_DATE(3)    <T.GROSS(3)>"
       "<D.PAY_DATE(4)    <T.GROSS(4)>"
       "<D.PAY_DATE(5)    <T.GROSS(5)>"
       "<D.PAY_DATE(6)    <T.GROSS(6)>"
    GOTO UPDATE
    ENDCASE

    CASE UPDATE
6.  REPEAT BUFFNUMBER
       MATCH PAY_DATE
          ON NOMATCH REJECT
          ON MATCH UPDATE GROSS
    ENDREPEAT
    GOTO COLLECT
    ENDCASE
    DATA
    END
```

The procedure processes as follows:

1. When the procedure is executed, the first CRTFORM is displayed:

```
PAY HISTORY UPDATE

ENTER EMPLOYEE ID #:
```

2. Assume the operator enters the following ID and transmits the data:

```
ENTER EMPLOYEE ID #: 071382660
```

   If there is a match, the instruction is to REPEAT the logic six times. That is, up until six times, find a PAY_DATE and hold the PAY_DATE and the GROSS in the HOLD buffer.

3. When there are no more PAY_DATE fields or six of them have been held, the procedure branches to CASE DISPLAY.

4. The procedure stores the number of records that are in the HOLD buffer in the variable BUFFNUMBER.

5. The procedure displays the following CRTFORM:

```
PAY HISTORY UPDATE

ENTER EMPLOYEE ID #: 071382660

FILL IN GROSS AMOUNT FOR EACH PAY DATE

PAY DATE           GROSS AMOUNT
820831             916.67
820730             916.67
820630             916.67
820528             916.67
820430             916.67
820331             916.67
```

The operator makes changes to the fields in the GROSS AMOUNT column and presses Enter. All changes for all records are transmitted simultaneously as shown:

```
PAY HISTORY UPDATE

ENTER EMPLOYEE ID #: 071382660

FILL IN GROSS AMOUNT FOR EACH PAY DATE

PAY DATE           GROSS AMOUNT
820831             816.67
820730             816.67
820630             816.67
820528             916.67
820430             916.67
820331             916.67
```

6. The REPEAT statement instructs FOCUS to perform the MODIFY logic on all segment instances.

Note: If a CRTFORM screen with subscripted variables is rejected with a FORMAT ERROR, you may not alter any records on the screen prior to the record rejected, as FOCUS has already held them.

Using Groups of Fields With Case Logic

When you use Case Logic to process a group of fields, some important rules apply:

- Each time the procedure enters the case, the next group of fields is processed. FOCUS keeps track internally of which groups have been processed.

- If the CRTFORM with the group of fields is not in the TOP case, you must create your own branching logic to process all the groups before going back to the TOP. This normally requires some kind of counting mechanism. Once the procedure goes back to the TOP case, all unprocessed data on the CRTFORM or in a lowercase is lost.

Example: Case Logic, Groups, CURSORINDEX and VALIDATE

In the following example, Case Logic is used with groups of fields. The CURSORINDEX (see *Specifying Cursor Position* on page 256) is used in conjunction with a VALIDATE:

```
MODIFY FILE EMPLOYEE
1.  CRTFORM
       "EMPLOYEE SALARY AND DEPARTMENT UPDATE"
       " "
       "PRESS ENTER"
    GOTO COLLECT

    CASE COLLECT
2.  REPEAT 6 TIMES
       NEXT EMP_ID
          ON NEXT HOLD EMP_ID CURR_SAL DEPARTMENT
          ON NONEXT GOTO DISPLAY
    ENDREPEAT
    GOTO DISPLAY
    ENDCASE

    CASE DISPLAY
3.  IF HOLDCOUNT EQ 0 GOTO EXIT;
4.  COMPUTE
       BUFFNUMBER/I5 = HOLDCOUNT;
5.  CRTFORM LINE 7
       "EMPLOYEE             SALARY               DEPARTMENT"
       "--------             ------               ----------"
       "<D.EMP_ID(1)         <:AAA.T.CSAL(1)>     <:BBB.T.DPT(1)>"
       "<D.EMP_ID(2)         <:AAA.T.CSAL(2)>     <:BBB.T.DPT(2)>"
       "<D.EMP_ID(3)         <:AAA.T.CSAL(3)>     <:BBB.T.DPT(3)>"
       "<D.EMP_ID(4)         <:AAA.T.CSAL(4)>     <:BBB.T.DPT(4)>"
       "<D.EMP_ID(5)         <:AAA.T.CSAL(5)>     <:BBB.T.DPT(5)>"
       "<D.EMP_ID(6)         <:AAA.T.CSAL(6)>     <:BBB.T.DPT(6)>"
```

```
6.  REPEAT 6 TIMES
       COMPUTE
          CURSOR/A66 = ':AAA';
          CURSORINDEX/I5=REPEATCOUNT;
       VALIDATE
          SALTEST = IF CSAL GT 50000 THEN 0 ELSE 1;
          ON INVALID TYPE "SALARY MUST BE LESS THAN $50,000"
          ON INVALID GOTO DISPLAY
    ENDREPEAT
    GOTO UPDATE
    ENDCASE

    CASE UPDATE
7.  REPEAT BUFFNUMBER
       MATCH EMP_ID
          ON NOMATCH REJECT
          ON MATCH UPDATE CURR_SAL DEPARTMENT
    ENDREPEAT
    GOTO COLLECT
    ENDCASE
    DATA
    END
```

The example processes as follows:

1. The first CRTFORM requests the operator to press Enter without typing anything.
2. The REPEAT statement retrieves six employee IDs, salaries, and department assignments and places them in a buffer.
3. If there are no records in the buffer, the procedure terminates.
4. The COMPUTE statement stores the number of records in the buffer in the variable BUFFNUMBER.
5. The second CRTFORM retrieves the IDs, salaries, and department assignments from the buffer and displays them together on the screen. Note the field labels:
   - ❏ The label :AAA on the CURR_SAL (CSAL) field.
   - ❏ The label :BBB on the DEPARTMENT (DPT) field.

Assume that the operator changes the values to the following:

```
EMPLOYEE SALARY AND DEPARTMENT UPDATE

PRESS ENTER

EMPLOYEE             SALARY               DEPARTMENT
--------             ------               ----------
071382660            35000.00             PRODUCTION
112847612            23200.00             MIS
117593129            75480.00             MIS
119265415            19500.00             PRODUCTION
119329144            39700.00             PRODUCTION
123764317            36862.00             PRODUCTION
```

6. The second REPEAT statement operates on each of the six records displayed by the second CRTFORM, in order of display, performing the following tasks:

   - Sets the CURSOR variable to the label :AAA.
   - Sets the CURSORINDEX variable to the number of the record it's processing (1 through 6).
   - Validates the CURR_SAL field value. If the CURR_SAL value is $50,000 or more, the procedure branches back to the beginning of Case DISPLAY. The procedure displays the second CRTFORM again, with the CURSOR and CURSORINDEX variables positioning the cursor on the invalid salary.

   In the example, the procedure positions the cursor on the third CURR_SAL value:

```
EMPLOYEE SALARY AND DEPARTMENT UPDATE

PRESS ENTER

EMPLOYEE             SALARY               DEPARTMENT
--------             ------               ----------
071382660            35000.00             PRODUCTION
112847612            23200.00             MIS
117593129            75480.00             MIS
119265415            19500.00             PRODUCTION
119329144            39700.00             PRODUCTION
123764317            36862.00             PRODUCTION

(FOC421)TRANS 2 REJECTED INVALID SALTEST
SALARY MUST BE LESS THAN $50,000
```

7. If all values are valid, the third REPEAT statement updates the employee's salary and department for each record in the buffer. The procedure then branches to Case COLLECT to update six more records in the data source.

Handling Errors

It is important to know how various errors are handled by FIDEL so that proper instructions can be given to terminal operators. The following errors can cause a transaction or screen of data to be rejected:

- A format error, caused by entering non-numeric data for a numeric field.
- A validation error, caused by entering an incoming value that failed a VALIDATE test coded in the MODIFY.
- A NOMATCH condition, caused by entering data for a key field that did not match any record in the data source.
- A DUPLICATE condition, caused by key field values that matched records on a data source.
- An ACCEPT error, caused by entering a value for a data source field that failed the ACCEPT test.

Note: Error messages are discussed in detail in *Messages: TYPE, LOG, and HELPMESSAGE* on page 130.

Handling Format Errors

If the operator enters a non-numeric character into a field defined as numeric, an error message is displayed and the screen is not processed (processing stops). The error message indicates the line number and field name in error and the cursor is automatically positioned on that field. Additionally, if the operator enters a value that fails an ACCEPT test for a field an error message is displayed and the screen is not processed. Any message specified for that field with the HELPMESSAGE attribute will also be displayed.

The operator can retype the data and press the Enter key to retransmit the screen. Alternatively, the operator may press the PF2 key to cancel the transaction. The error prevents anything on the screen from being processed. When the operator corrects the error and transmits the screen, processing resumes.

There are two exceptions to this rule. When there are repeating groups, all complete transactions up to the error will be processed. Also, in REPEAT/HOLD loops, the data prior to the format error may not be altered.

VALIDATE and CRTFORM Display Logic

When the operator enters a value that is invalid, the transaction is rejected and an error message is displayed. By default, control returns to the first CRTFORM in the TOP case. However, you can use an ON INVALID GOTO statement to transfer control to any other case in the request.

If the NOCLEAR or blank option in the CRTFORM statement (see *Additional Screen Control Options* on page 293) is in effect, the screen will not be cleared. The operator can change the data in the offending transaction and retransmit the screen.

When you use validations, you can divide the tests into different cases and repeat a case if it fails the test. The advantage of this is that the operator can change the invalid data and retransmit the screen before other sections are processed. An ON INVALID TYPE phrase can be used to send an informative message to the operator on the screen. The following example shows the use of these options:

```
CASE TRY
CRTFORM
   EMPLOYEE ID #: <EMP_ID NAME: <LAST_NAME"
   "CURRENT SALARY: <CURR_SAL"
VALIDATE
   GOODSAL= CURR_SAL GT 10000 AND CURR_SAL LT 1000000;
   ON INVALID TYPE
   THE CURRENT SALARY CANNOT BE LARGER THAN 1000000 OR"
   "LESS THAN 10000"
   ON INVALID GOTO TRY
   .
   .
   .
```

All messages appear on the bottom four lines of the screen, unless you specify the TYPE option on the CRTFORM statement (see *Additional Screen Control Options* on page 293).

Handling Errors With Repeating Groups

If old style repeating groups (those without subscripts) are present and there is an error, processing continues to the next transaction on the screen. This means that if the operator changes the offending transaction and retransmits the screen, the other transactions on the screen become duplicates. It is important when using repeating groups to reject duplicates and turn the duplicate message off (LOG DUPL MSG OFF).

Alternatively, avoid using VALIDATE with repeating groups. Use COMPUTE instead and branch to a case that displays the erroneous data in a lower portion of the screen.

The following is an example of this technique. A test field is computed in Case TEST, using DECODE. This test field checks that the department value is a valid one. If the operator inputs a department value that is invalid, control branches to a case that displays the erroneous data (CASE BADDPT).

```
    MODIFY FILE EMPLOYEE
1.  CRTFORM
       "FILL IN THE FOLLOWING CHART WITH THE SALARIES"
       "AND DEPARTMENT ASSIGNMENTS OF FOUR NEW EMPLOYEES"
       " "
       "           EMPLOYEE ID       DEPARTMENT       SALARY"
       "           -----------       ----------       ------"
       "PERSON 1   <EMP_ID           <DEPARTMENT      <CURR_SAL"
       "PERSON 2   <EMP_ID           <DEPARTMENT      <CURR_SAL"
       "PERSON 3   <EMP_ID           <DEPARTMENT      <CURR_SAL"
       "PERSON 4   <EMP_ID           <DEPARTMENT      <CURR_SAL"
       GOTTO TEST

2.  CASE TEST
    IF EMP_ID IS ' ' GOTO TOP;
    COMPUTE
       TEST/I1 = DECODE DEPARTMENT (MIS 1 PRODUCTION 1 ELSE 0);
    IF TEST IS 0 GOTO BADDEPT ELSE GOTO ADD;
    ENDCASE

3.  CASE ADD
    MATCH EMP_ID
       ON NOMATCH INCLUDE
       ON MATCH REJECT
    ENDCASE

4.  CASE BADDEPT
    COMPUTE
       XEMP/A9 = EMP_ID;
       XDEPT/A10 = DEPARTMENT;
    CRTFORM LINE 12
       "INVALID ENTRY: DEPARTMENT MUST BE MIS OR PRODUCTION"
       "CORRECT THE ENTRY BELOW"
       " "
       "EMPLOYEE ID:  <D.XEMP    DEPARTMENT:  <T.XDEPT"
    COMPUTE
       DEPARTMENT=XDEPT;
    GOTO TEST
    ENDCASE

    DATA
    END
```

The request processes as follows:

1. This is the first and TOP case, and contains a CRTFORM that displays four instances of repeating groups. Assume the operator fills in values and transmits the screen. Control transfers to Case TEST.

2. Case TEST contains a computed field that uses DECODE to make sure that the values that have been input for DEPARTMENT are either MIS or PRODUCTION. When a DEPARTMENT value does not match this list, TEST is returned a code of 0, in which case control transfers to Case BADDEPT.
3. Case BADDEPT first computes two fields, XEMP and XDEPT, to have the values of EMP_ID and DEPARTMENT at the time the error occurred. Next, BADDEPT displays a CRTFORM containing a message to the operator and the two computed fields. The XDEPT field, which contains the invalid DEPARTMENT value, is a turnaround field so that the operator can see the invalid value and change it. Next, the COMPUTE is reversed and the new values are returned to their respective fields. Control transfers back to Case TEST where the DEPARTMENT values will continue to be tested until they are all valid. At that point, control transfers to Case ADD.
4. Case ADD contains the MATCH logic necessary to include new employees into the EMPLOYEE data source. The transaction including all the repeating groups is processed at one time.

Rejecting NOMATCH or Duplicate Data

When the request directs that transactions be rejected, an error message is displayed on the screen. It is a good idea, when the major keys do not repeat, to use a split CRTFORM and give the keys in the first CRTFORM. Once the keys are accepted, the rest of the data may be entered. For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "ENTER EMPLOYEE ID#:     <EMP_ID"
   "THEN PRESS ENTER"
MATCH EMP_ID
   ON NOMATCH TYPE
      "ID NOT IN DATABASE   PLEASE REENTER"
   ON NOMATCH REJECT
   ON MATCH CRTFORM LINE 4
      "LAST NAME:   <T.LAST_NAME"
      "DEPARTMENT:  <T.DEPARTMENT"
      "SALARY:      <T.CURR_SAL"
   ON MATCH UPDATE LAST_NAME DEPARTMENT CURR_SAL
DATA
END
```

If the EMP_ID does not match, control returns immediately to the operator with a request to correct the value. If a match does occur, the operator must then fill in the balance of the form and transmit it.

If repeating groups are present and no other cases are involved, all of the groups are processed before control returns to the screen. Thus, splitting screens in this way is particularly useful when the second CRTFORM contains repeating groups.

Logging Transactions

You can log the data entered on the screen to any log file. Only the data is logged, not the CRTFORM, so a compact log file is created. For example:

```
LOG TRANS ON ALLDATA
```

This will log transactions to a file allocated to the ddname ALLDATA.

The record length of the file must allow space for each field on each CRTFORM in the procedure, plus one character at the start of each CRTFORM. The record length should not be longer than this.

This may be an inconvenient format, since it is very long if several CRTFORMs exist. Instead you can construct a custom log file of your own design using TYPE statements. This example logs data collected from its preceding CRTFORM to a file allocated to ddname MYCRT, including a COMPUTE transaction number, TNUM:

```
CRTFORM
"EMPLOYEE ID #: <EMP_ID  NAME <LAST_NAME"
"HIRE DATE: <HIRE_DATE"
COMPUTE
TNUM/I4=TNUM+1;
TYPE ON MYCRT
"<TNUM><EMP_ID><LAST_NAME><HIRE_DATE"
```

This option is preferable to the standard LOG option whenever a procedure contains more than two CRTFORMs, or when text or computed fields must be captured on the log file.

Additional Screen Control Options

MODIFY CRTFORMs support several additional screen control options:

- Clearing the screen with *Clearing the Screen: CLEAR/NOCLEAR* on page 293.
- Specifying the screen size with *Specifying Screen Size: WIDTH/HEIGHT* on page 294.
- Changing the size of the message area at the bottom of the screen using *Changing the Size of the Message Area: TYPE* on page 296. This increases the length of the screen that can be used for the actual form.

Clearing the Screen: CLEAR/NOCLEAR

Data is transmitted from the CRTFORM to the data source when the operator presses the Enter key. After each successful screen is processed, the data areas are automatically cleared. You can override this default by using the NOCLEAR option. Then, after each data transmission, the screen remains unchanged.

This is a useful feature when there is a substantial amount of data that carries over from one screen to another. The syntax is

```
CRTFORM action
```

where:

action
Is one of the following:

blank is the default. Causes the screen to clear after the data is transmitted. If a transaction is invalid and an error message appears at the bottom of the screen, the screen will not be cleared.

NOCLEAR causes the data values on the screen to remain as is after data is transmitted.

CLEAR causes the data values on the screen to clear after every data transmission, even if there is an error. Thus, if CLEAR is specifically used and there is an error, data must be reentered.

Note: The options chosen may be different from one CRTFORM to the next.

Specifying Screen Size: WIDTH/HEIGHT

FIDEL assumes a default screen size of 24 lines of 80 characters each. The WIDTH/HEIGHT options allow you to use the full width and height of IBM terminals that are larger than the usual 3270 screen for the display of CRTFORMs. The following syntax allows you to override the defaults

```
CRTFORM [WIDTH nnn] [HEIGHT nnn]
```

where:

WIDTH *nnn*

Is the total number of characters across the face of the screen. Acceptable values for WIDTH are 80 and 132 and cannot exceed the true width of the terminal. FOCUS verifies that each line of the CRTFORM can be displayed at the current WIDTH specification. If any line of the CRTFORM exceeds it, you will receive error message FOC456, and the procedure will not run.

`HEIGHT` *nnn*

Is the total number of lines that each screen supports. It bears no relation to the number of lines in the CRTFORM. It may not exceed the true height of the terminal, but it may be less. For example, you can specify HEIGHT 20 for a Model 2 screen instead of 24 and write a CRTFORM of 32 lines. The first 16 lines appear on one screen and the next 16 on the subsequent screen. Remember that by default, four lines are reserved for TYPE messages.

The following table gives the physical screen sizes for the IBM 3270 series of terminals:

Terminal Type	Model	Width	Height
3270	1	80	24
3277, 3278, 3279, 3178	2	80	24
3278, 3279	3	80	32
3278	4	80	43
3278	5	132	27

FOCUS senses the width and height of the terminal which you are using and attempts to implement your CRTFORM WIDTH and HEIGHT specifications accordingly. Here are some rules and facts that apply:

- If your WIDTH or HEIGHT specifications exceed the perceived characteristics of the terminal, you will receive a FOC491 error message and the procedure will not run.
- FOCUS sees the terminal as it is defined to the operating system. For example, a Model 5 3278 may be defined to the operating system as a Model 2 terminal. That terminal will appear to FOCUS as a Model 2 (24 lines deep and 80 characters wide). A WIDTH 132 specification will produce a FOC491 error message.

Changing the Size of the Message Area: TYPE

By default, FOCUS reserves the last four lines of the terminal screen for TYPE messages and text messages specified with the HELPMESSAGE attribute (see *Messages: TYPE, LOG, and HELPMESSAGE* on page 130). You can override this default and determine the number of lines each CRTFORM reserves with the keyword TYPE. This feature allows you to increase the number of lines on the screen for CRTFORM display and reduce the number of lines reserved for messages at the bottom of the screen. The syntax is

```
CRTFORM TYPE {n|4}
```

where:

n

Is a number from one to four indicating the number of message lines desired. The TYPE value setting remains in effect for all subsequent CRTFORMs in the same procedure until overridden by a new value.

You can expand the actual CRTFORM screen size by specifying a number less than four. For example, a terminal with a height of 24 lines currently reserves 20 lines for the CRTFORM and four lines for the TYPE area. If you specify a TYPE area of 2, the CRTFORM area increases to 22 lines.

If one procedure varies the size of the TYPE area from a larger to a smaller number, CRTFORM will clear the necessary TYPE statements in order to generate the next screen. If multiple CRTFORMs are written to the same screen, each CRTFORM should specify the same TYPE area size. For example:

```
CRTFORM LINE 1 TYPE 2
:
:
CRTFORM LINE 7 TYPE 2
```

Messages supplied with the HELPMESSAGE attribute in the Master File for fields on the MODIFY CRTFORM, are displayed in the TYPE area.

This type of message consists of one line of text which is displayed when:

- The value entered for a data source field is invalid according to the ACCEPT test for the field, or causes a format error.
- The user places the cursor in the data entry area for a particular field and presses a predefined PF key. If no message has been specified for that field, the following message will be displayed:

```
NO HELP AVAILABLE FOR THIS FIELD
```

Using FIDEL in Dialogue Manager

FIDEL works with all the standard Dialogue Manager facilities. However, the following differences apply when you use FIDEL with Dialogue Manager:

- You must allocate space for the variable field on the -CRTFORM, because variable fields in Dialogue Manager are not related to a Master File (see *Allocating Space on the Screen for Variable Fields* on page 297).
- There are two additional control statements: -CRTFORM BEGIN and -CRTFORM END. These give you control over when you begin and end the form (see *Starting and Ending CRTFORMS: BEGIN/END* on page 298). This control allows you to make use of other Dialogue Manager control statements as you are building your -CRTFORM.

Allocating Space on the Screen for Variable Fields

You must define the length of variable fields in -CRTFORMs. The length of Dialogue Manager variables can be defined in one of two ways:

- Directly on the -CRTFORM using the following syntax for allocating space.

  ```
  <&variable/ length
  ```

  where:

  length

  Is a number representing the alphanumeric length of the variable.

- By using the -SET command to pre-declare the allocation of space using the syntax

  ```
  -SET &variable = ' ' ;
  ```

  where:

  ' '

  Represents the alphanumeric length of the variable.

  Note:

  - If the variable format has been previously defined in the FOCEXEC procedure, the length defined directly on the -CRTFORM supersedes the previously defined format permanently.

- Variables used as label names (&:variable) cannot be automatically defined on the -CRTFORM. These variables must be defined with -SET statements.

Starting and Ending CRTFORMS: BEGIN/END

-CRTFORM BEGIN indicates that the form is being built. This Dialogue Manager control statement enables you to use other Dialogue Manager control statements between the screen lines without causing the CRTFORM to end. This is necessary when you are using indexed variables in a looping procedure.

-CRTFORM END terminates the form and causes the display of the assembled form.

Example: Using Indexed Variables With -CRTFORM BEGIN and -CRTFORM END

The following is an example of the use of indexed variables in -CRTFORM. The variable &LINENUM is the indexed variable in the -CRTFORM. The index, &I, is set to increment by 1 each time a line is written. After the 10th line, the -CRTFORM ends. Note the use of the Dialogue Manager label, -BUILD and the -SET statement to control the loop within the form:

```
1.  -SET &I = 0;
2.  -CRTFORM BEGIN
    -"THE FOLLOWING FORM STORES 10 LINES OF TEXT"
    -" "
3.  -BUILD
4.  -SET &I = &I + 1;
5.  -SET &LINENUM.&I = 'LINE ' | &I;
6.  -"<D.&LINENUM.&I <&LINE.&I/60"
7.  -IF &I LT 10 GOTO BUILD;
8.  -CRTFORM END
    -*
    -TYPE LINE #2 CONTAINS THE FOLLOWING TEXT:
    -TYPE
9.  -TYPE &LINE2
```

This example processes as follows:

1. This -SET statement declares a counter, &I, and sets the counter to 0.
2. The -CRTFORM BEGIN statement begins the form.
3. This statement is a Dialogue Manager label, -BUILD. Because we are using the -CRTFORM BEGIN statement, this label does not end the CRTFORM.
4. This -SET statement sets the counter &I to increment by 1 each time a line is written. This controls the loop within the form.
5. This -SET statement indexes the variable &LINENUM with the counter &I. Thus, each time it is encountered in the -CRTFORM it will increment +1.

6. The -CRTFORM will appear as follows:

```
THE FOLLOWING FORM STORES 10 LINES OF TEXT

LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
LINE 6
LINE 7
LINE 8
LINE 9
LINE 10
```

   Type any text you wish onto the lines.

7. The -IF test allows the loop to process until there are 10 lines in the -CRTFORM. At that point control transfers to the -CRTFORM END statement.
8. -CRTFORM END ends the -CRTFORM and causes it to be displayed.
9. The last TYPE statement shows the contents of line 2.

Clearing the Screen in Dialogue Manager

The statement -CRTFORM both initiates the screen form and automatically clears the screen. The screen form begins at the top of the screen.

After the operator enters values for the variables and presses *Enter*, the variables are supplied with the values and the screen is cleared.

Changing the Size of the Message Area: -CRTFORM TYPE

By default, FOCUS reserves the last four lines of the Dialogue Manager terminal screen for TYPE messages. You can change this by using the keyword TYPE to determine the number of lines each CRTFORM reserves for messages. This feature allows you to increase the number of lines on the screen for CRTFORM display and reduce the number of lines reserved for messages at the bottom of the screen. The syntax is

```
-CRTFORM TYPE {n|4}
```

where:

n

Is a number from 1 to 4 indicating the number of message lines desired. The TYPE value setting remains in effect for all subsequent CRTFORMs in the same procedure until overridden by a new value. The default is 4.

You can expand the CRTFORM screen size by specifying a number less than 4. For example, a terminal with a height of 24 lines reserves 20 lines for the CRTFORM and four lines for the TYPE area. If you specify a TYPE area of 2, the CRTFORM area increases to 22 lines.

Annotated Example: -CRTFORM

The following FOCEXEC is an example of a TABLE request incorporating the use of -CRTFORM.

```
    -* Component Of Retail Sales Reporting Module
1.  SET &LIST = 'STAMFORD,UNIONDALE,NEWARK';
2.  PROMPT &CITY.(&LIST).ENTER CITY.:
    -*
3.  -CRTFORM
    -"Monthly Sales Report For <D.&CITY"
    -"Date: <D.&DATE     Time: <D.&TOD"
    -" "
    -"Beginning Product Code is:  <&BEGCODE/3"
    -"Ending Product Code is:     <&ENDCODE/3"
    -"Regional Supervisor is:     <&REGIONMGR/15"
    -"Title For UNIT_SOLD is:     <&UNIT_HEAD/10"

4.  TABLE FILE SALES
    HEADING CENTER
    MONTHLY REPORT FOR &CITY"
    "PRODUCT CODES FROM &BEGCODE TO &ENDCODE"
    SUM UNIT_SOLD AS &UNIT_HEAD
    AND RETURNS AND COMPUTE
    RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
    BY PROD_CODE
    IF PROD_CODE IS-FROM &BEGCODE TO &ENDCODE
    IF CITY EQ &CITY
    FOOTING CENTER
    "REGION MANAGER: &REGIONMGR"
    "CALCULATED AS OF &DATE"
5.  END
```

The following is a sample of the dialogue between the screen and the operator. Operator entries are in lowercase.

1. The -SET statement sets a value for the variable &LIST. The value is actually a list of the names of three cities. They are enclosed in single quotation marks because of the embedded commas.

2. The -PROMPT statement prompts the operator at the terminal for a value for &CITY. Assume the operator types a city that is not on the list:

```
ENTER CITY:
boston
PLEASE CHOOSE ONE OF THE FOLLOWING:
STAMFORD,UNIONDALE,NEWARK
ENTER CITY:
stamford
```

3. The statement -CRTFORM initiates a screen form on which you type data:

```
Monthly Sales Report for STAMFORD
Date: 01/08/2003              Time: 13.12.41

Beginning Product Code is:    b10
Ending Product Code is:       b20
Regional Supervisor is:       smith
Title For UNIT_SOLD is:       sales
```

4. The following are the stacked FOCUS commands as they appear on the FOCSTACK after the values have been entered from the -CRTFORM:

```
TABLE FILE SALES
HEADING CENTER
"MONTHLY REPORT FOR STAMFORD"
"PRODUCT CODES FROM B10 TO B20"
" "
SUM UNIT_SOLD AS SALES AND RETURNS AND COMPUTE
RATIO/D5.2 = 100 * RETURNS/UNIT_SOLD;
BY PROD_CODE
IF PROD_CODE IS-FROM B10 TO B20
IF CITY EQ STAMFORD
FOOTING CENTER
"REGION MANAGER: SMITH"
"CALCULATED AS OF 01/08/2003"
END
```

5. The report is as follows:

```
PAGE     1

  MONTHLY REPORT FOR STAMFORD
 PRODUCT CODES FROM B10 TO B20

PROD_CODE  SALES   RETURNS  RATIO
---------  ------  -------  -----
B10            60       10  16.67
B12            40        3   7.50
B17            29        2   6.90

             REGION MANAGER: SMITH
              CALCULATED AS OF 11/04/03
```

Using the FOCUS Screen Painter

The FOCUS Screen Painter allows you to design a FIDEL full-screen layout by placing literal text and areas for fields on the screen in any position that you desire. You then assign these field areas of the screen to a data source or computed fields, and FOCUS automatically codes the CRTFORM. You can also color, highlight, and/or assign screen attributes to sections of the screen (text, fields, background or any combination).

The FOCUS Screen Painter also allows you to generate CRTFORMs automatically without specifying field names (see *Generating Automatic CRTFORMs* on page 270).

The Screen Painter operates within TED, the FOCUS editor (see the *Overview and Operating Environments* manual for more details on TED), and can be used to create both MODIFY CRTFORMs and Dialogue Manager -CRTFORMs. It is easy to use and makes the creating of forms simple and visual.

Entering Screen Painter

To create a CRTFORM using the Screen Painter, you first enter the PAINT command from within TED. You can set up the PAINT screen as follows:

1. Enter TED by typing TED followed by the name of the file:

   ```
   TED FOCEXEC(CRTEMP
   ```

   This opens the FOCEXEC called CRTEMP. The FOCEXEC may or may not already exist.

2. Place a CRTFORM or -CRTFORM statement in the FOCEXEC if it is not already there. For example:

   ```
   MODIFY FILE EMPLOYEE
   CRTFORM
   ```

3. When a FOCEXEC is on the screen, enter the PAINT command in the command area or press PF4. TED searches from the current line down the file until it finds a CRTFORM statement and makes the following line the current line. (If you use more than one CRTFORM in the FOCEXEC and you want to create the second CRTFORM, enter the command PAINT 2.)

   Note: A Master File must be active for the Screen Painter to set the default field lengths for data source fields.

   The following PAINT screen is displayed on your terminal:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7..+...




...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:_

01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
```

Between the two scale lines are 20 blank lines in which to enter the screen layout. The cursor is positioned in the command zone in the lower left portion of the screen. The codes at the bottom of the screen identify some of the PF keys that you can use.

These perform the following functions:

PF Key	Function
01=HELP	Lists all the PF key functions.
03=END	Transfers you from the PAINT screen back into TED, within your file.
07=BACKWARD	Scrolls back to the previous screen of the CRTFORM. When used with ASSIGN, moves the cursor back to the first field.
08=FORWARD	Scrolls forward to the next screen of the CRTFORM. When used with ASSIGN, moves the cursor to the next field.

PF Key	Function
09=ASGN-FLD	Use on the ASSIGN screen. Transfers you to the particular field that the cursor is placed on. You can then immediately assign or change attributes for that field.
10=ASSIGN	Transfers you from the PAINT screen to the ASSIGN screen (see *Identifying Fields: ASSIGN* on page 310).
11=FIDEL	Shows you the CRTFORM as it will appear on the screen.
17=BOX	Enables you to define a box of text. Move the cursor to the upper-left corner and press PF17. Select features from the box menu and then move the cursor to the bottom-right corner and press PF17.

Note: With the exception of FORWARD, BACKWARD and ASGN-FLD, you can also accomplish these functions by typing the command name in the command zone.

4. If the CRTFORM already includes fields, and one or more fields are not declared in the Master File, you may see this message:

   ```
   (FOC532) LENGTHS OF FIELDS IN THIS CRTFORM CANNOT BE DETERMINED
   ```

   To continue type IGNore and provide the lengths explicitly, or type ?F filename to activate the appropriate master. After you follow the message instructions, the PAINT screen appears.

PF Keys in PAINT

You can alter the values of PF keys in PAINT with the command

```
SET PFnnword
```

where:

nn

Is a number from 1 to 24 specifying the PF key to be set.

word

Is the new value for the key.

The initial PF key settings in PAINT are:

PF Key	Setting
PF1, PF13	: HELP
PF2, PF14	: INSERT
PF3, PF15	: END
PF4	: PAINT
PF5	: TOP
PF6	: BOTTOM
PF7, PF19	: BACKWARD PAGE
PF8, PF20	: FORWARD PAGE
PF9	: ASSIGN FIELD
PF10	: ASSIGN
PF11	: FIDEL
PF12	: DUPLICATE
PF16	: QUIT
PF17	: BOX
PF18	: (currently not used)
PF21	: CRTFORM
PF22	: SET OUTPUT FIDEL
PF23	: SET OUTPUT DIALOGUE
PF24	: (currently not used)

Entering Data Onto the Screen

In PAINT, you may enter text, and specify field dimensions. Always use the arrow keys to designate text and field areas on this screen. Generally, text is entered by positioning the cursor and typing, but fields require type and width specifications.

To create a field, type

```
<xx...x
```

where the total number of x's equals the width of the field desired. If you do not specify a width, or if the command you entered is not syntactically correct, or active, PAINT will automatically default to a width defined in the Master File.

Fields are conditional by default. To specify non-conditional fields, enter

```
<xx...x>
```

where the total number of x's equals the width of the field.

You may enter text descriptions of each field, but do not type the field name after the left or right caret. Later you will learn how to assign each field a field name. You may designate the field as Entry, Turnaround or Display with the ASSIGN command (see *Identifying Fields: ASSIGN* on page 310). By default, the fields are conditional. To specify non-conditional, type a right caret (>) after the x's that indicate the field. We recommend that turnaround fields be non-conditional. (See *Conditional and Non-Conditional Fields* on page 264 for information on conditional and non-conditional fields.)

Editing Functions

When you are designing your screen, you have editing functions available to you. To use them, you must enter the command name on the COMMAND line on your PAINT screen or use the appropriate PF key:

- Inserting Lines: INSERT, PF2, PF14. You can insert lines by moving the cursor to any character on a line. Press *PF2* or *PF14* and the new line will be inserted immediately following the line where the cursor is positioned. If you want to insert more than one line, type the command (do not press *Enter*)

  ```
  I[NSERT] n
  ```

  where *n* is the number of new lines to be inserted. Next, move the cursor to the line where you want the lines inserted. Press *Enter* and *n* lines will be inserted beneath the line where the cursor is currently positioned.

  If the insert causes the screen to exceed 20 lines, the message

```
1,40
```

will be displayed, indicating that the display starts at line 1 out of a total of 40.

- Deleting Lines: DELETE. You can similarly delete lines by typing:

  ```
  D[ELETE] n
  ```

  on the command line, where *n* is the number of lines you want deleted. Next, move the cursor to the first line you want deleted and press *Enter*.

- Duplicating Lines: DUPLICATE, PF12. You can duplicate lines by placing the cursor on the line that you want to duplicate. Press *PF12*. If you want to duplicate more than one line, type the command

  ```
  DU[PLICATE] n
  ```

  where *n* is the number of copies you want; position the cursor on the line you want to duplicate and press *Enter*.

  If the line that you are copying contains subscripted fields (for example, "SALES (1)"), the subscripts will be incremented by one automatically (see *Specifying Groups of Fields* on page 281). If you want an increment other than 1, enter the command

  ```
  DUPLICATE n m
  ```

  where *m* is the increment number.

Sample PAINT Screen

In the following example, assume that the following FOCEXEC exists:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "ENTER EMPLOYEE ID #: <EMP_ID"
MATCH EMP_ID
   ON NOMATCH REJECT
   ON MATCH CRTFORM
```

To use the Screen Painter to create the second CRTFORM, specify PAINT 2 at the TED command line (2 indicates second CRTFORM). Then type the following text and fields on the PAINT screen to create the CRTFORM that will be displayed if there is a match on EMP_ID.

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: <XXXXXXXXX            LAST NAME: <XXXXXXXXXXXXXXX

DEPARTMENT: <XXXXXXXXXX>             CURRENT SALARY: <XXXXXXXX

BANK: <XXXXXXXXXXXXXXXXXXXX



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:_

01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
```

When you finish entering text and indicating areas for fields (the number of X's corresponds to the field length), press *Enter*. The following screen results:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: <111111111          LAST NAME: <22222222222222

DEPARTMENT: <1111111111>           CURRENT SALARY: <22222222

BANK: <11111111111111111111



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:_

01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
```

Note that the X's are replaced with numbers indicating the relative position of each field on a line. On the second line, EMPLOYEE ID is number 1 and LAST NAME is number 2.

Note: Labels created in Screen Painter cannot exceed 12 characters.

Defining a Box on the Screen

You can define a boxed area of the screen, have it flash, or underline it. Text within the box assumes the attributes of the box, but fields within the box do not change their appearance.

To define a box, place the cursor in the upper-left corner of the area you want to enclose in a box, and press *PF17*. The following screen and menu appear:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: <111111111           LAST NAME: <22222222222222

DEPARTMENT: <1111111111>            CURRENT SALARY: <22222222

BANK: <11111111111111111111



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
Color (W,B,R,P,G,A,Y):          Flash /Under/Inv/Off (F,U,I,O):
Please position the cursor at other end of box and hit the key again
```

Fill in the color and/or attributes that you desire, position the cursor at the lower-right corner of the area you want to enclose in a box, and press *PF17*.

To delete the box, move the cursor to the upper-left corner of the box and type O in the attribute area. Then move the cursor to the lower-right corner of the box and press *PF17*. The letter O stands for OFF and deletes the box. Note that you must position the cursor exactly at the corners.

The BOX feature of Screen Painter will not generate a proper box if the fields cross or touch the boundary of the box itself. Boxes may not extend past column 77.

If you try to generate a box, but fail, the following message appears:

```
command.box
(FOC694) INVALID BOX REGION OR CURSOR POSITION DEFINED.
```

When this happens, press *Enter* to clear the message, move the cursor to the upper-left corner, and press *PF17* to start over.

If you press *PF17* to begin a box and then decide not to define a box, press *PF3* to cancel.

Identifying Fields: ASSIGN

Until now, you have simply laid out text that describes the fields, designated a display length (X's) within the left caret (<), and possibly indicated non-conditional (>) fields. Now you can assign field names and attributes for the fields. Enter the command ASSIGN in the command zone or press *PF10*. Your ASSIGN screen displays the following:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: *********            LAST NAME: EEEEEEEEEEEEEEE

DEPARTMENT: EEEEEEEEEE             CURRENT SALARY: EEEEEEEE

BANK: EEEEEEEEEEEEEEEEEEE



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
Field:               Entry/Turn Disp (E,T,D):    Col (W,B,R,P,G,A,Y):
Field Length: 9(D12.2M)  High/Nodis/Inv (H,N,I):     Label:
```

The first field following the descriptive text EMPLOYEE ID #: is highlighted and replaced by asterisks. All other fields are displayed in low intensity with E's denoting the length of the fields. The cursor is positioned in the status entry area at the bottom of the screen next to FIELD.

Now you can enter and assign field names and attributes for the field appearing in asterisks. Fill in the appropriate values in the status entry area at the bottom of the screen. To move from one status area to the next, press *TAB*. You may leave a blank where you do not want to use a particular attribute.

`FIELD:`

Enter the field name for the first field. In this case, enter EMP_ID, which is the name of the field in the Master File.

`ENTRY/TURN/DISP (E,T,D):`

You may designate the field as Entry, Turnaround, or Display by specifying E, T, or D, respectively. The default is Entry. (See *Data Entry, Display and Turnaround Fields* on page 239 for more information on Entry, Turnaround, and Display fields.) You specify whether a field is conditional or non-conditional when you enter the field on the PAINT screen (see *Entering Data Onto the Screen* on page 306).

`COL (W,B,R,P,G,A,Y):`

You may designate the field with a color by entering one of the color abbreviations in the COL area. You may choose W, white; B, blue; R, red; P, pink; G, green; A, aqua; Y, yellow. If you do not wish to assign a color, leave this area blank.

`FIELD LENGTH: 9 (A9):`

In MODIFY, if a Master File is active while you are assigning attributes, the LENGTH status will contain two values: the first value is the number of X's from the PAINT screen, which is the display value; the value in parentheses is the format value from the Master File. The display value must be equal to or less than the format value.

If you want to change the display value on the screen, put a new number in the FIELD LENGTH area or return to PAINT (PF3) and enter the correct number of characters following the <.

`HIGH/NODISP/INV (H,N,I):`

You can choose highlight, nodisplay or inverse video as an attribute for the field by filling in the appropriate abbreviation.

`LABEL:`

If you want to enter a label, simply enter its name. The colon and period are automatically provided on the screen.

In the following example, the current field is LAST_NAME. It is designated a display field. The remaining attributes are left blank. After you press *Enter* and move to the next field, the asterisks turn to D's (display) as did the EMP_ID field.

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: DDDDDDDDD           LAST NAME: ***************

DEPARTMENT: EEEEEEEEEE             CURRENT SALARY: EEEEEEEEEEEEEE

BANK: EEEEEEEEEEEEEEEEEEE



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
FIELD: last_name      ENTRY/TURN/DISP (E,T,D): d    COL (B,R,P,G,A,Y):
FIELD LENGTH: 15 (A,15) HIGH/NODISP/INV (H,N,I):       LABEL:
```

To move to the next field, press *PF8*. You may assign a field name, prefix, color, attribute or label to the remaining fields on the screen. If you need to move to a previous field to change something, press *PF7*. This will return you to the first field. From there you can use the TAB key to move to the field that you need.

To move to a specific field directly from PAINT or from within ASSIGN, place the cursor on that field and press *PF9*, ASGN-FLD.

Viewing the Screen: FIDEL

From the PAINT or ASSIGN screen, you can view the exact FIDEL screen that you have created. Press *PF11* or type FIDEL in the command zone. As the following screen shows, all entry fields are blank and ready to receive data; all turnaround fields contain T's and may be typed over; all display fields contain D's and are protected:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

                                EMPLOYEE UPDATE

EMPLOYEE ID #: DDDDDDDDD           LAST NAME: DDDDDDDDDDDDDDD

DEPARTMENT: TTTTTTTTTT             CURRENT SALARY:



FIDEL: Press PF3 or PF15 to return to the PAINT screen.
```

As indicated on the FIDEL screen, to return to the PAINT screen press *PF3* or *PF15*.

Generating CRTFORMs Automatically

To generate CRTFORMs automatically (that is, without specifying individual fields) from the FOCUS Screen Painter, use the asterisk (*) with CRTFORM in the PAINT screen command zone. (See *Generating Automatic CRTFORMs* on page 270 for information on CRTFORM * variations and syntax.)

The text description identifying field is the field name from the Master File. Key fields automatically become entry fields, and all other fields become turnaround fields. With multi-segment data sources, the CRTFORM * command ignores all segments following the first cross-reference (segment type KU or KM) described in the Master File.

For example, to generate a CRTFORM containing all fields in the EMPLOYEE Master File, do the following:

1. Type a MODIFY and a CRTFORM statement in a FOCEXEC.
2. Enter PAINT on the TED command line to invoke the Screen Painter.
3. Type CRTFORM * in the Screen Painter command zone.

The following PAINT screen results:

```
...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...

EMP_ID          :<111111111>   :
LAST_NAME       :<111111111111111   :        FIRST NAME  :<2222222222:
HIRE_DATE       :<111111    :                DEPARTMENT  :<2222222222:
CURR_SAL        :<111111111111  :            CUR_JOBCODE :<222   :
ED_HRS          :<111111    :
BANK_NAME       :<11111111111111111111  :
BANK_CODE       :<111111    :                BANK_ACCT   :<222222222:
EFFECT_DATE     :<111111    :
DAT_INC         :<111111>   :
PCT_INC         :<111111    :                SALARY      :<222222222222:
JOBCODE         :<111      :
TYPE            :<1111>    :
ADDRESS_LN1     :<11111111111111111111  :
ADDRESS_LN2     :<11111111111111111111  :
ADDRESS_LN3     :<11111111111111111111  :
ACCTNUMBER      :<111111111  :
PAY_DATE        :<111111>   :
GROSS           :<111111111111   :
DED_CODE        :<1111>   :
PF8=NEXT SCREEN PF7=PREVIOUS SCREEN PF1=OUT



...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
COMMAND:                                                 1, 40
01=HELP 03=END 07=BACKWARD 08=FORWARD 09=ASGN-FLD 10=ASSIGN 11=FIDEL 17=BOX
```

CRTFORM * creates labels (that is, text describing each field) on the CRTFORM of up to 12 characters. If the field name is shorter than 12 characters, the label is the field name. If the field name exceeds 12 characters, a caret (>) in the 12th position indicates a longer field name.

Terminating Screen Painter

To return to TED from the PAINT screen, enter the command END in the command zone or press PF3 until the prompt for TED appears. TED displays the lines as they have been generated, beginning at the current line, which is ON MATCH CRTFORM:

```
" <.C.                      EMPLOYEE UPDATE                              <0X
" <.C.                                                                   <0X
                          <.C."
" <.C. EMPLOYEE ID #: <D.EMP_ID/09                    LAST NAME:        <0X
<LAST_NAME/15             <.C."
" <.C.                                                                   <0X
                          <.C."
" <.C. DEPARTMENT: <T.DEPARTMENT/10>             CURRENT SALARY:        <0X
<T.CURR_SAL/08            <.C."
" <.C.                                                                   <0X
                          <.C."
" <.C. BANK <T.BANK_NAME/20                                           <.C."
" <.C.                                                                   <0X
DATA
END
```

The generated code for the CRTFORM is in the file. Notice that each field is named and has its length appended to it. Any attributes or labels requested during the ASSIGN process are also present. If you want to change the layout, you can use the TED editor or you can return to the PAINT and/or ASSIGN screen to make the changes.

You can add further MATCH logic to the FOCEXEC by using TED. For example:

```
MODIFY FILE EMPLOYEE
CRTFORM
   "ENTER EMPLOYEE ID #: <EMP_ID"
   MATCH EMP_ID
      ON NOMATCH REJECT
      ON MATCH CRTFORM
"        EMPLOYEE UPDATE"
" "
" EMPLOYEE ID #: <D.EMP_ID/09            LAST NAME: <D.LAST_NAME/15"
" "
" DEPARTMENT: <:FIRST.H.T.DEPARTMENT/10> CURRENT SALARY: <0X
<.C.CURR_SAL/08"
" "
" BANK : <BANK_NAME/20"
      ON MATCH UPDATE DEPARTMENT CURR_SAL
      ON MATCH CONTINUE TO BANK_NAME
      ON NOMATCH INCLUDE
      ON MATCH REJECT
DATA
END
```

If you want to add another CRTFORM screen at this point, make sure you are on the current line, type the CRTFORM or -CRTFORM statement, and reenter PAINT to design the next screen. Finally, you can exit the PAINT screen, return to TED, and add or change further logic.

Alternatively, all of the logic of the request could have been entered first and then the Screen Painter used to create all the FIDEL screens. To create the first screen, enter the command PAINT or PAINT 1; to create the second screen, enter the command PAINT 2. PAINT 2 locates the second CRTFORM statement starting from the current line. You can continue with PAINT 3, and so on, for all subsequent CRTFORM statements in the procedure.

Chapter 3

Creating and Rebuilding a Data Source

You can create a new data source, or re-initialize an existing data source, using the CREATE command.

After a data source exists, you may find it necessary to reorganize it in order to use disk space more effectively, to change the contents, index, or structure of the data source, or to change legacy date fields to smart date fields. You can do all of this and more using the REBUILD command.

You can use the CREATE and REBUILD commands with FOCUS and XFOCUS data sources. You can also use the CREATE command to create relational tables for which you have the appropriate data adapter.

In the remainder of this chapter, all references to FOCUS data sources apply to FOCUS and XFOCUS data sources.

In this chapter:

Creating a New Data Source: The CREATE Command

You can create a new, empty FOCUS data source for a Master File using the CREATE command. You can also use the CREATE command to erase the data in an existing FOCUS data source.

The CREATE command also works, with the appropriate data adapter installed, for a relational table (such as a DB2 or Teradata table). For information, see the documentation for the relevant data adapter.

If you issue the CREATE FILE command when the data source already exists, the following message appears for a FOCUS or XFOCUS data source:

```
(FOC441) WARNING. THE FILE EXISTS ALREADY. CREATE WILL WRITE OVER IT.
REPLY:
```

The DROP option on the CREATE FILE command prevents the display of the message and creates the data source, dropping the existing table first, if necessary, and re-parsing the Master File if it changed.

Note that you must issue either an allocation or a CREATE command for a new data source. For all other platforms, if the data source has not been initialized, a CREATE is automatically issued on the first MODIFY or Maintain Data request made against the data source.

Syntax: How to Use the CREATE Command

```
CREATE FILE mastername [DROP]
```

where:

mastername

Is the name of the Master File that describes the data source.

DROP

Drops an existing file before performing the CREATE and re-parses the Master File, if necessary. No warning messages are generated.

If you issue the CREATE FILE *filename* DROP command for a FOCUS or XFOCUS data source that has an external index or MDI, you must REBUILD the index after creating the data source.

Note the following when issuing CREATE:

- If you do not allocate the data source prior to issuing the CREATE command, the data source is created as a temporary data set. To retain the data source, copy it to a permanent data set with the DYNAM COPY command.

- The CREATE command preformats the primary space allocation and initializes the data source entry in the File Directory Table. A Master File must exist for the data source in a PDS allocated to ddname MASTER.
- Issuing MODIFY or Maintain commands against data sources for which no CREATE or allocation was issued results in a read error.

After you enter the CREATE command, the following appears:

```
NEW FILE name ON date AT time
```

where:

name

Is the complete name of the new data source.

ON *date* AT *time*

Is the date and time at which the data source was created or recreated.

When you issue the CREATE command without the DROP option, if the data source already exists, the following message appears:

```
(FOC441) WARNING. THE FILE EXISTS ALREADY. CREATE WILL WRITE OVER IT.
REPLY:
```

To erase the data source and create a new, empty data source, enter Y. To cancel the command and leave the data source intact, enter END, QUIT, or N.

If you wish to give the data source absolute File Integrity protection, issue the following command prior to the CREATE command:

```
SET SHADOW=ON
```

Example: Creating a FOCUS Data Source

To create the ADDRESS data source, allocate the data source and then issue the CREATE command:

```
DYNAM ALLOC F(ADDRESS) DA(ADDRESS.FOCUS) NEW SPACE(5,5) CYL
 CREATE FILE ADDRESS
```

The following message displays:

```
NEW FILE ADDRESS            ON 03/02/1999 AT 15.16.59
```

This creates the new FOCUS data source ADDRESS.FOCUS allocated to ddname ADDRESS.

Rebuilding a Data Source: The REBUILD Command

You can make a structural change to a FOCUS data source after it has been created using the REBUILD command. Using REBUILD and one of its subcommands REBUILD, REORG, INDEX, EXTERNAL INDEX, CHECK, TIMESTAMP, DATE NEW, and MDINDEX, you can:

- Rebuild a disorganized data source (REBUILD).
- Delete instances according to a set of screening conditions (REBUILD or REORG).
- Redesign an existing data source. This includes adding and removing segments, adding and removing data fields, indexing different fields, changing the size of alphanumeric data fields and more (REORG).
- Index new fields after rebuilding or creating the data source (INDEX).
- Create an external index database that facilitates indexed retrieval when joining or locating records (EXTERNAL INDEX).
- Check the structural integrity of the data source (CHECK). Check when the FOCUS data source was last changed (TIMESTAMP).
- Convert legacy date formats to smart date formats (DATE NEW).
- Build or modify a multi-dimensional index (MDINDEX).

You can use the REBUILD facility:

- **Interactively at the screen,** by issuing the REBUILD command at the FOCUS command prompt.
- **As a batch procedure,** by entering the REBUILD command, the desired subcommand, and any responses to subcommand prompts on separate lines of a procedure.

Before using the REBUILD facility, you should be aware of several required and recommended prerequisites regarding file allocation, security authorization, and backup.

Reference: Before You Use REBUILD: Prerequisites

Before you use the REBUILD facility, there are several prerequisites that you must consider:

- **Partitioning.** You can only REBUILD one partition of a partitioned FOCUS data source at one time. You must explicitly allocate the partition you want to REBUILD. Alternatively, you can create separate Master Files for each partition.

- **Size.** To REBUILD a FOCUS data source that is larger than one-gigabyte you must explicitly allocate ddname REBUILD to a temporary file with enough space to contain the data. It is strongly recommended that you REBUILD/REORG to a new file, in sections, to avoid the need to allocate large amounts of space to REBUILD. In the DUMP phase, use selection criteria to dump a section of the data source. In the LOAD phase, make sure to add each new section after the first. To add to a data source you must issue the LOAD command with the following syntax:

  ```
  LOAD NOCREATE
  ```

- **Allocation.** Usually, you do not have to allocate workspace prior to using a REBUILD command. It is automatically allocated. However, adequate workspace must be available. As a rule of thumb, have space 10 to 20% larger than the size of the existing file available for the REBUILD and REORG options.

  The file name REBUILD is always assigned to the workspace. In the DUMP phase of the REORG command, the allocation statement appears in case you want to perform the LOAD phase at a different time.

- **Security authorization.** If the data source you are rebuilding is protected by a database administrator, you must be authorized for read and write access in order to perform any REBUILD activity.

- **Backup.** Although it is not a requirement, we recommend that you create a backup copy of the original Master File and data source before using any of the REBUILD subcommands.

Procedure: How to Use the REBUILD Facility

The following steps describe how to use the REBUILD facility:

1. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

2. Select a subcommand by supplying its name or its number. The following list shows the subcommand names and their corresponding numbers:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smart date formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

Your subsequent responses depend on the subcommand you select. Generally, you will only need to give the name of the data source and possibly one or two other items of information.

If you are using the REBUILD facility interactively, you must allocate SYSPRINT to the terminal in order to view the menu. For more information on using SYSPRINT, see the *Overview and Operating Environments* manual.

Note: If you select the wrong subcommand interactively, you can enter QUIT to exit.

Controlling the Frequency of REBUILD Messages

When REBUILD processes a data source, it displays status messages periodically (for example, REFERENCE..AT SEGMENT 1000) to inform you of the progress of the rebuild. The default display interval is every 1000 segment instances processed during REBUILD retrieval and load phases. The number of messages that appear is determined by the number of segment instances in the FOCUS data source being rebuilt, divided by the display interval.

Syntax: How to Control the Frequency of REBUILD Messages

REBUILD displays a message (REFERENCE..AT SEGMENT *segnum*) at periodic intervals to inform you of its progress as it processes a data source. You can control the frequency with which REBUILD displays this message by issuing the command

```
SET REBUILDMSG = {n|1000}
```

where:

n

Is any integer from 1,000 to 99,999,999 or 0 (to disable the messages).

A setting of less than 1000:

- Generates a warning message that describes the valid values (0 or greater than 999).
- Keeps the current setting. The current setting will either be the default of 1000, or the last valid integer greater than 999 to which REBUILDMSG was set.

Example: Controlling the Display of REBUILD Messages

The following messages are generated for a REBUILD CHECK where REBUILDMSG has been set to 4000, and the data source contains 19,753 records.

```
STARTING..
REFERENCE..AT SEGMENT    4000
REFERENCE..AT SEGMENT    8000
REFERENCE..AT SEGMENT   12000
REFERENCE..AT SEGMENT   16000
NUMBER OF SEGMENTS RETRIEVED=   19753
CHECK COMPLETED...
```

Optimizing File Size: The REBUILD Subcommand

You use the REBUILD subcommand for one of two reasons. Primarily, you use it to improve data access time and storage efficiency. After many deletions, the physical structure of your data does not match the logical structure. REBUILD REBUILD dumps data into a temporary work space and then reloads it, putting instances back in their proper logical order. A second use of REBUILD REBUILD is to delete segment instances according to a set of screening conditions.

Normally, you use the REBUILD subcommand as a way of maintaining a clean data source. To check if you need to rebuild your data source, enter the ? FILE command (described in *Confirming Structural Integrity Using ? FILE and TABLEF* on page 340):

```
? FILE filename
```

If your data source is disorganized, the following message appears:

```
FILE APPEARS TO NEED THE -REBUILD-UTILITY
REORG PERCENT IS A MEASURE OF FILE DISORGANIZATION
0 PCT IS PERFECT -- 100 PCT IS BAD
REORG PERCENT x%
```

This message appears whenever the REORG PERCENT measure is more than 30%. The REORG PERCENT measure indicates the degree to which the physical placement of data in the data source differs from its logical, or apparent, placement.

The &FOCDISORG variable can be used immediately after the ? FILE command and also shows the percentage of disorganization in a data source. &FOCDISORG will show a data source percentage of disorganization even if it is below 30% (see the *Developing Applications* manual).

Procedure: How to Use the REBUILD Subcommand

The following steps describe how to use the REBUILD subcommand:

1. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

   The following options are available:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smartdate formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

2. Select the REBUILD subcommand by entering:

   ```
   REBUILD or 1
   ```

3. Enter the name of the data source to be rebuilt.

   On z/OS, enter Enter the ddname.

4. If you are simply rebuilding the data source and require no selection tests, enter:

   ```
   NO
   ```

   The REBUILD procedure will begin immediately.

   On the other hand, if you wish to place screening conditions on the REBUILD subcommand, enter:

   ```
   YES
   ```

   Then enter the necessary selection tests, ending the last line with ,$.

   Test relations of EQ, NE, LE, GE, LT, GT, CO (contains), and OM (omits) are permitted. Tests are connected with the word AND, and lists of literals may be connected with the OR operator. A comma followed by a dollar sign (,$) is required to terminate any test.

   For example, you might enter the following:

   ```
   A EQ A1 OR A2 AND B LT 100 AND
   C GT 400 AND D CO 'CUR',$
   ```

Statistics appear when the REBUILD REBUILD procedure is complete, including the number of segments retrieved and the number of segments included in the rebuilt data source.

Using the REBUILD Subcommand

The following examples illustrate how to use the REBUILD subcommand.

Example: Using the REBUILD Subcommand

The following example illustrates using the REBUILD subcommand interactively.

```
rebuild

Enter option
1. REBUILD       (Optimize the database structure)
2. REORG         (Alter the database structure)
3. INDEX         (Build/modify the database index)
4. EXTERNAL INDEX(Build/modify an external index database)
5. CHECK         (Check the database structure)
6. TIMESTAMP     (Change the database timestamp)
7. DATE NEW      (Convert old date formats to smartdate formats)
8. MDINDEX       (Build/modify a multidimensional index)
rebuild

ENTER NAME OF FOCUS/FUSION FILE
> employee

ANY RECORD SELECTION TESTS? (YES/NO)
> no
 STARTING..
  DCB USED WITH FILE REBUILD  IS DCB=(RECFM=VB,LRECL=00088,BLKSIZE=23940)
 NUMBER OF SEGMENTS RETRIEVED=     576
  NEW FILE EMPLOYEE           ON 05/14/1999 AT 09.31.26
 NUMBER OF SEGMENTS INPUT=     576
 FILE HAS BEEN REBUILT
```

Changing Data Source Structure: The REORG Subcommand

The REORG subcommand enables you to make a variety of changes to the Master File after data has been entered in the FOCUS data source. REBUILD REORG is a two-step procedure that first dumps the data into a temporary workspace and then reloads it under a new Master File.

You can use REBUILD REORG to:

- Add new segments as descendants of existing segments.
- Remove segments.
- Add new data fields as descendants to an existing segment.

  Note: The fields must be added after the key fields.
- Remove data fields.
- Change the order of non-key data fields within a segment. Key fields may not be changed.
- Promote fields from unique segments to parent segments.

- Demote fields from parent segments to descendant unique segments.
- Index different fields or remove indexes.
- Increase or decrease the size of an alphanumeric data field.

REBUILD REORG will not enable you to:

- Change field format types (alphanumeric to numeric and vice versa, changing numeric format types).
- Change the value for SEGNAME attributes.
- Change the value for SEGTYPE attributes.
- Change field names that are indexed.

To accomplish these tasks you must use FIXFORM. See your MODIFY, documentation for more information.

Procedure: How to Use the REORG Subcommand

The following steps describe how to use the REORG subcommand:

1. Before making any changes to the original Master File, make a copy of it with another name.
2. Using an editor, make the desired edits to the copy of the Master File.
3. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

   The following options are available:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smartdate formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

4. Select the REORG subcommand by entering:

   ```
   REORG or 2
   ```

   The options are:

   ```
   1. DUMP           (DUMP contents of the database)
   2. LOAD           (LOAD data into the database)
   ```

If you want to mount a scratch tape for work space during the DUMP phase, you can type the name of the tape after the word REORG.

5. Initiate the DUMP phase of the procedure by entering:

```
DUMP or 1
```

6. Enter the name of the data source you wish to dump from. Be sure to use the name of the original Master File for this phase.

   On z/OS, enter Enter the ddname.

7. You can specify selection tests by entering YES. Only data that meets your specifications will be dumped. It is more likely, however, that you will want to dump the entire data source. To do so, enter:

```
NO
```

   Statistics appear during the DUMP procedure, including the number of segments dumped and the name and statistics for the temporary file used to hold the data.

8. After the DUMP phase is complete, you are ready to begin the second phase of REBUILD REORG: LOAD. Enter:

```
REBUILD
```

9. Select the REORG subcommand by entering:

```
REORG or 2
```

   The options are:

```
1. DUMP     (DUMP contents of the database)
2. LOAD     (LOAD data into the database)
```

10. Initiate the LOAD phase of the procedure by entering:

```
LOAD or 2
```

11. Enter the name of the data source you wish to load from the temporary file created during the dump phase. In most cases, this will be the new data source name.

At this stage, you have loaded the specified data from the original Master File into a new data source with the name you specified. It is important to remember that both the Master File and data source for the original Master File remain. You have three choices:

- You may want to rename the original Master File and data source to prevent possible confusion.
- You may rename the new Master File and data source to the original name. As a result, any existing FOCEXECs referencing the original name will run against the new data source.

- You may delete the original Master File and data source after you verify that the new Master File and data source are correct and complete.

If you enter the name of a data source that already exists, (the original Master File) you are prompted that you will be appending data to a preexisting data source and asked if you wish to continue.

You are not asked if you want to append to an existing data source. The data source is created. If you want to append, when you issue the LOAD command, enter LOAD NOCREATE.

Enter N to terminate REBUILD execution. Enter Y to add the records in the temporary REBUILD file to the original FOCUS data source.

If duplicate field names occur in a Master File, REBUILD REORG is not supported.

You must issue either an allocation or a CREATE for a new data source being loaded.

Using the REORG Subcommand

The following examples illustrate how to use the REORG subcommand.

Example: Using the REORG Subcommand

First make a copy of the data source:

```
dynam copy employee.focus oldemp.focus
```

Now start the DUMP phase:

```
rebuild
```

```
Enter option
 1. REBUILD         (Optimize the database structure)
 2. REORG           (Alter the database structure)
 3. INDEX           (Build/modify the database index)
 4. EXTERNAL INDEX  (Build/modify an external index database)
 5. CHECK           (Check the database structure)
 6. TIMESTAMP       (Change the database timestamp)
 7. DATE NEW        (Convert old date formats to smartdate formats)
 8. MDINDEX         (Build/modify a multidimensional index)
reorg
Enter option
 1. DUMP              (DUMP contents of the database)
 2. LOAD              (LOAD data into the database)

dump

DUMP
ENTER NAME OF FOCUS/FUSION FILE
 > employee
ANY RECORD SELECTION TESTS? (YES/NO)
> no
 STARTING..
  DCB USED WITH FILE REBUILD  IS DCB=(RECFM=VB,LRECL=00088,BLKSIZE=23940)
 NUMBER OF SEGMENTS RETRIEVED=     576
```

Now start the LOAD phase:

```
 >  >  rebuild
 Enter option
  1. REBUILD          (Optimize the database structure)
  2. REORG            (Alter the database structure)
  3. INDEX            (Build/modify the database index)
  4. EXTERNAL INDEX   (Build/modify an external index database)
  5. CHECK            (Check the database structure)
  6. TIMESTAMP        (Change the database timestamp)
  7. DATE NEW         (Convert old date formats to smartdate formats)
  8. MDINDEX          (Build/modify a multidimensional index)
 >   reorg|
 Enter option
  1. DUMP             (DUMP contents of the database)
  2. LOAD             (LOAD data into the database)
LOAD
ENTER NAME OF FOCUS/FUSION FILE
> employee

 STARTING..
  NEW FILE EMPLOYEE          ON 05/14/1999 AT 09.41.37
 NUMBER OF SEGMENTS INPUT=     576
 >  >
```

Indexing Fields: The INDEX Subcommand

To index a field after you have entered data into the data source, use the INDEX subcommand. You can index fields in addition to those previously specified in the Master File or since the last REBUILD or CREATE command. The only requirement is that each field specified must be described with the FIELDTYPE=I (or INDEX=I) attribute in the Master File.

The INDEX option uses the operating system sort program. You must have disk space to which you can write. To calculate the amount of space needed, add 8 to the length of the index field in bytes and multiply the sum by twice the number of segment instances

```
(LENGTH + 8) * 2n
```

where:

n

Is the number of segment instances.

You may decide to wait until after loading data to add the FIELDTYPE=I attribute and index the field. This is because the separate processes of loading data and indexing can be faster than performing both processes at the same time when creating the data source. This is especially true for large data sources.

Sort libraries and workspace must be available. The REBUILD allocates default sort work space if you have not already. DDNAMEs SORTIN and SORTOUT must be allocated prior to issuing a REBUILD INDEX.

Procedure: How to Use the INDEX Subcommand

The following steps describe how to use the INDEX subcommand:

1. Add the FIELDTYPE=I attribute to the field or fields you are indexing in the Master File.
2. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

   The following options are available:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smartdate formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

3. Select the INDEX subcommand by entering:

   ```
   INDEX or 3
   ```

4. Enter the name of the Master File in which you will add the FIELDTYPE=I or INDEX=I attribute.
5. Enter the name of the field to index. If you are indexing all the fields that have FIELDTYPE=I, enter an asterisk (*).

Statistics appear when the REBUILD INDEX procedure is complete, including the field names that were indexed and the number of index values included.

Using the INDEX Subcommand

The following examples illustrate how to use the INDEX subcommand.

Example: Using the INDEX Subcommand

REBUILD INDEX uses an external sort. FOCUS searches for the system-installed sort product using its normal search path.

```
>  > tso alloc f(sortin) sp(1 1) tracks
>  > tso alloc f(sortout) sp(1 1) tracks
>  > tso alloc f(sysout) da(*)
>  > rebuild

Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX             (Build/modify the database index)
 4. EXTERNAL INDEX    (Build/modify an external index database)
 5. CHECK             (Check the database structure)
 6. TIMESTAMP         (Change the database timestamp)
 7. DATE NEW          (Convert old date formats to smartdate formats)
 8. MDINDEX           (Build/modify a multidimensional index)
> 3
INDEX

ENTER THE NAME OF THE MASTER
> employee
ENTER NAME OF FIELD TO INDEX (OR * FOR ALL)
> emp_id
STARTING..
(FOC319) WARNING. THE FIELD IS INDEXED AFTER THE FILE WAS CREATED:
EMP_ID
INDEX VALUES RETRIEVED=      12
SORT COMPLETE .. RET CODE     0
INDEX INITIALIZED FOR: EMP_ID
INDEX VALUES INCLUDED=       12
```

Creating an External Index: The EXTERNAL INDEX Subcommand

Users with READ access to a local FOCUS data source can create an index database that facilitates indexed retrieval when joining or locating records. An external index is a FOCUS data source that contains index, field, and segment information for one or more specified FOCUS data sources. The external index is independent of its associated FOCUS data source. External indexes offer equivalent performance to permanent indexes for retrieval and analysis operations.

External indexes enable indexing on concatenated FOCUS data sources, indexing on real and defined fields, and indexing selected records from WHERE/IF tests. External indexes are created as temporary data sets unless preallocated to a permanent data set. They are not updated as the indexed data changes.

You create an external index with the REBUILD command. Internally, REBUILD begins a process which reads the databases that make up the index, gathers the index information, and creates an index database containing all field, format, segment, and location information.

You provide information about:

- Whether you want to add new records from a concatenated database to the index database.
- The name of the external index database that you want to build.
- The name of the data source from which the index information is obtained.
- The name of the field from which the index is to be created.
- Whether you want to position the index field within a particular segment.
- Any valid WHERE or IF record selection tests.

Sort libraries and work space must be available. The REBUILD allocates default sort work space if you have not already. DDNAMEs SORTIN and SORTOUT must be allocated prior to issuing a REBUILD.

Procedure: How to Use the EXTERNAL INDEX Subcommand

To create an external index from a concatenated database, follow these steps:

1. Assume that you have the following USE in effect:

```
USE CLEAR *
USE
EMPLOYEE
EMP2 AS EMPLOYEE
JOBFILE
EDUCFILE
END
```

   Note that EMPLOYEE and EMP2 are concatenated and can be described by the EMPLOYEE Master File.

2. Initiate the REBUILD facility by entering:

```
REBUILD
```

The following options are available:

```
1. REBUILD        (Optimize the database structure)
2. REORG          (Alter the database structure)
3. INDEX          (Build/modify the database index)
4. EXTERNAL INDEX (Build/modify an external index database)
5. CHECK          (Check the database structure)
6. TIMESTAMP      (Change the database timestamp)
7. DATE NEW       (Convert old date formats to smartdate formats)
8. MDINDEX        (Build/modify a multidimensional index)
```

3. Select the EXTERNAL INDEX subcommand by entering:

   ```
   EXTERNAL INDEX or 4
   ```

4. Specify whether to create a new index data source or add to an existing one by entering one of the following choices:

   ```
   NEW
   ADD
   ```

   For this example, assume you are creating a new index database and respond by entering:

   ```
   NEW
   ```

5. Specify the name of the external index database:

   ```
   EMPIDX
   ```

6. Specify the name of the data source from which the index records are obtained:

   ```
   EMPLOYEE
   ```

7. Specify the name of the field to index:

   ```
   CURR_JOBCODE
   ```

8. Specify whether the index should be associated with a particular field by entering YES or NO. For this example, enter:

   ```
   NO
   ```

9. Indicate whether you require any record selection tests by entering YES or NO.

   For this example, enter:

   ```
   NO
   ```

If you responded YES, you would next enter the record selection tests, ending them with the END command on a separate line.

For example:

```
IF DEPARTMENT EQ 'MIS'
END
```

You will see statistics (output of the ? FDT query) about the index data source when the REBUILD EXTERNAL INDEX procedure is complete. This query is automatically issued at the end of the REBUILD EXTERNAL INDEX process in order to validate the contents of the index database.

Example: **External Index Statistics**

The following illustrates external index statistics.

```
EXTERNAL INDEX FILE:                EMPIDX
FULL NAME:                          EMPIDX.FOCUS
VERSION :
DATE/TIME OF LAST CHANGE:           05/13/99 15.40.46

   EXTERNAL INDEX DATABASE PAGES:   00000001
   DATABASE INDEXED:                EMPLOYEE
   FIELD NAME:                      EMPINFO.CURR
   FIELD FORMAT:                    A3
   SEGMENT NAME:                    EMPINFO
   SEGMENT LOCATION:                EMPLOYEE

EXTERNAL INDEX DATA COMPONENTS:
   EMPLOYEE.FOCUS
   EMP2.FOCUS
```

Reference: **Special Considerations for REBUILD EXTERNAL INDEX**

Consider the following when working with external indexes:

- Up to eight indexes can be activated at one time in a USE list using the WITH statement. More than eight indexes may be activated in a session if you issue the USE CLEAR command and issue new USE statements.
- Up to 256 concatenated files may be indexed. However, only eight indexes may be activated at one time.

- Verification of the component files is now done for both the date and time stamp of file creation. Files with the same date and time stamp that are copied display the following message:

  ```
  (FOC995) ERROR. EXTERNAL INDEX DUPLICATE COMPONENT: fn REBUILD ABORTED
  ```

- MODIFY may only use the external index with the FIND or LOOKUP functions. The external index cannot be used as an entry point, such as:

  ```
  MODIFY FILE filename.indexfld
  ```

- Indexes may not be created on field names longer than twelve characters.
- Text fields may not be used as indexed fields.
- The USE options NEW, READ, ON, LOCAL, and AS *master* ON *userid* READ are not supported for the external index database.
- The external index database need not be allocated since CREATE FILE automatically performs a temporary allocation. If a permanent database is required, then an allocation for the index database must be in place prior to the REBUILD EXTERNAL INDEX command.
- SORTIN and SORTOUT, work files that the REBUILD EXTERNAL INDEX process creates, must be allocated with adequate space. In order to estimate the space needed, the following formula may be used:

  ```
  bytes = (field_length + 20) * number_of_occurrences
  ```

Concatenating Index Databases

The external index feature enables indexed retrieval from concatenated FOCUS data sources. If you wish to concatenate databases that comprise the index, you must issue the appropriate USE command prior to the REBUILD. The USE must include all cross-referenced and LOCATION files. REBUILD EXTERNAL INDEX contains an add function that enables you to append only new index records from a concatenated database to the index database, eliminating the need to recreate the index database.

The original data source from which the index was built may not be in the USE list when you add index records. If it is, REBUILD EXTERNAL INDEX generates the following message:

```
(FOC999) WARNING. EXTERNAL INDEX COMPONENT REUSED: ddname
```

Positioning Indexed Fields

The external index feature is useful for positioning retrieval of indexed values for defined fields within a particular segment in order to enhance retrieval performance. By entering at a lower segment within the hierarchy, data retrieved for the indexed field is affected, as the index field is associated with data outside its source segment. This enables the creation of a relationship between the source and target segments. The source segment is defined as the segment that contains the indexed field. The target segment is defined as any segment above or below the source segment within its path.

If the target segment is not within the same path, the following message is generated:

```
(FOC974) EXTERNAL INDEX ERROR. INVALID TARGET SEGMENT
```

A defined field may not be positioned at a higher segment.

While the source segment can be a cross-referenced or LOCATION segment, the target segment cannot be a cross-referenced segment. If an attempt is made to place the target on a cross-referenced segment, the following message is generated:

```
(FOC1000) INVALID USE OF CROSS REFERENCE FIELD
```

If you choose not to associate your index with a particular field, the source and target segments will be the same.

Activating an External Index

After building an external index database, you must associate it with the data sources from which it was created. This is accomplished with the USE command. The syntax is the same as when USE is issued prior to building the external index database, except the WITH or INDEX option is required.

Syntax: How to Activate an External Index

```
USE  [ADD|REPLACE]
database_name [AS mastername]
index_database_name  [WITH|INDEX]  mastername   .
   .
   .
END
```

where:

ADD

Appends one or more new databases to the present USE list. Without the ADD option, the existing USE list is cleared and replaced by the current list of USE databases.

`REPLACE`

Replaces an existing *database_name* in the USE list.

`database_name`

Is the name of the data source.

On z/OS, enter Enter the ddname.

You must include a data source name in the USE list for all cross-referenced and LOCATION files that are specified in the Master File.

`AS`

Is used with a Master File name to concatenate data sources.

`mastername`

Specifies the Master File.

`index_database_name`

Is the name of the external index database.

On z/OS, enter Enter the ddname.

`WITH|INDEX`

Is a keyword that creates the relationship between the component data sources and the index database. INDEX is a synonym for WITH.

Checking Data Source Integrity: The CHECK Subcommand

It is rare for the structural integrity of a FOCUS data source to be damaged. Structural damage will occasionally occur, however, during a drive failure or if an incorrect Master File is used. In this situation, the REBUILD CHECK command performs two essential tasks:

- It checks pointers in the data source.
- Should it encounter an error, it displays a message and attempts to branch around the offending segment or instance.

Although CHECK is able to report on a good deal of data that would otherwise be lost, it is important to remember that frequently backing up your FOCUS data sources is the best method of preventing data loss.

CHECK will occasionally fail to uncover structural damage. If you have reason to believe that there is damage to your data source, though CHECK reports otherwise, there is a second method of checking data source integrity. This method entails using the ? FILE and TABLEF commands. Though this is not a REBUILD function, it is included at the end of this section because of its relevancy to CHECK.

Procedure: How to Use the CHECK Subcommand

The following steps describe how to use the CHECK subcommand:

1. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

   The following options are available:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smartdate formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

2. Select the CHECK subcommand by entering:

   ```
   CHECK or 5
   ```

3. Enter the name of the data source to be checked.

   On z/OS, enter Enter the ddname.

Statistics appear during the REBUILD CHECK procedure:

- If no errors are found, the statistics indicate the number of segments retrieved.
- If errors are found, the statistics indicate the type and location of each error:

  DELETE indicates that the data has been deleted and should not have been retrieved.

  OFFPAGE indicates that the address of the data is not on a page owned by this segment.

  INVALID indicates that the type of linkage cannot be identified. It may be a destroyed portion of the data source.

Using the CHECK Option

The following examples illustrate how to use the CHECK option.

Example: Using the Check Option (File Undamaged)

The following example illustrates using the CHECK option interactively.

```
rebuild
Enter option
 1. REBUILD           (Optimize the database structure)
 2. REORG             (Alter the database structure)
 3. INDEX             (Build/modify the database index)
 4. EXTERNAL INDEX    (Build/modify an external index database)
 5. CHECK             (Check the database structure)
 6. TIMESTAMP         (Change the database timestamp)
 7. DATE NEW          (Convert old date formats to smartdate formats)
 8. MDINDEX           (Build/modify a multidimensional index)
> 5
CHECK
ENTER NAME OF FOCUS/FUSION FILE
>
> employee
STARTING..
NUMBER OF SEGMENTS RETRIEVED=     576
CHECK COMPLETED...
>  >
```

Confirming Structural Integrity Using ? FILE and TABLEF

When you believe that there is damage to your data source, though REBUILD CHECK reports there is not, use the ? FILE and TABLEF commands to compare the number of segment instances reported after invoking each command. A disparity indicates a structural problem.

Procedure: How to Verify REBUILD CHECK Using ? FILE and TABLEF

1. Issue the following command:

   ```
   ? FILE filename
   ```

   where:

   filename

   Is the name of the FOCUS data source you are examining.

   A report displays information on the status of the data source. The number of instances for each segment is listed in the ACTIVE COUNT column.

2. To ensure that the TABLEF command in the next step counts all segment instances, including those in the short paths, issue the command:

   ```
   SET ALL = ON
   ```

3. Enter:

```
TABLEF FILE filenameCOUNT field1 field2END
```

where:

filename

Is the name of the Master File of the FOCUS data source.

field1...

Are the names of fields in the data source. Name one field from each segment. It does not matter which field is named in the segment.

The report produced shows the number of field occurrences for those fields named and thus the number of segment instances for each segment. These numbers should match their respective segment instance numbers shown in the ? FILE command (except for unique segments which the TABLEF command shows to have as many instances in the parent segment). If the numbers do not match, or if either the ? FILE command or TABLEF command ends abnormally, the data source is probably damaged.

Example: Checking the Integrity of the EMPLOYEE Data Source

User input is shown in bold. Computer responses are in uppercase:

```
? FILE
STATUS OF FOCUS FILE: EMPLOYEE   ON 01/31/2003 AT 16.17.32
                ACTIVE  DELETED    DATE OF    TIME OF    LAST TRANS
SEGNAME         COUNT   COUNT      LAST CHG   LAST CHG   NUMBER

EMPINFO           12              05/13/1999   16.17.22      448
FUNDTRAN           6              05/13/1999   16.17.22       12
PAYINFO           19              05/13/1999   16.17.22       19
ADDRESS           21              05/13/1999   16.17.22       21
SALINFO           70              05/13/1999   16.17.22      448
DEDUCT           448              05/13/1999   16.17.22      448
TOTAL SEGS       576
TOTAL CHAR      8984
TOTAL PAGES        8
LAST CHANGE                       05/13/1999   16.17.22      448
SET ALL = ON
TABLEF FILE EMPLOYEE
COUNT EMP_ID BANK_NAME DAT_INC TYPE PAY_DATE DED_CODE
END

PAGE     1

 EMP_ID  BANK_NAME  DAT_INC  TYPE   PAY_DATE  DED_CODE
 COUNT   COUNT      COUNT    COUNT  COUNT     COUNT
 ------  ---------  -------  -----  --------  --------
     12         12       19     21        70       448
NUMBER OF RECORDS IN TABLE=      488 LINES= 1
```

Note that the BANK_NAME count in the TABLEF report is different than the number of FUNDTRAN instances reported by the ? FILE query. This is because FUNDTRAN is a unique segment and is always considered present as an extension of its parent.

Changing the Data Source Creation Date and Time: The TIMESTAMP Subcommand

A FOCUS data source date and time stamp are updated each time the data source is changed by SCAN, FSCAN, CREATE, REBUILD, HLI, Maintain, or MODIFY. You can update a data source date and time stamp without making changes to the data source by using REBUILD TIMESTAMP subcommand.

Procedure: How to Use the TIMESTAMP Subcommand

The following steps describe how to use the TIMESTAMP subcommand:

1. Initiate the REBUILD facility by entering:

   ```
   REBUILD
   ```

   The following options are available:

   ```
   1. REBUILD        (Optimize the database structure)
   2. REORG          (Alter the database structure)
   3. INDEX          (Build/modify the database index)
   4. EXTERNAL INDEX (Build/modify an external index database)
   5. CHECK          (Check the database structure)
   6. TIMESTAMP      (Change the database timestamp)
   7. DATE NEW       (Convert old date formats to smartdate formats)
   8. MDINDEX        (Build/modify a multidimensional index)
   ```

2. Select the TIMESTAMP subcommand by entering:

   ```
   TIMESTAMP or 6
   ```

3. Enter the name of the data source whose date and time stamp is to be updated.

   On z/OS, enter Enter the ddname.

4. Enter one of the following options for the source of the date and time:

   T (today's date). Updates the data source date and time stamp with the current date and time.

   D (search file for date). Updates the data source date and time stamp with the last date and time at which the data source was actually changed. Each page of the data source is scanned and the most recent date and time recorded for a page is applied to the data source. This is the same as issuing the ? FILE query, and can be time consuming when the data source is very large. This option is used to keep an external index database synchronized with its component data source.

   MMDDYY HHMMSS. Is a date and time that you specify, which REBUILD will use to update the data source date and time stamp. The date and time that you enter must have the format mmddyy hhmmss or mmddyyyy hhmmss. There must be a space between the date and the time. If you use two digits for the year, REBUILD uses the values for DEFCENT and YRTHRESH to determine the century.

   If you supply an invalid date or time, the following message appears:

   ```
   (FOC961) INVALID DATE INPUT IN REBUILD TIME:
   ```

Converting Legacy Dates: The DATE NEW Subcommand

The REBUILD subcommand DATE NEW converts legacy dates (alphanumeric, integer, and packed-decimal fields with date display options) to smart dates (fields in date format) in your FOCUS data sources.

The utility uses update-in-place technology. It updates your data source and creates a new Master File, yet does not change the structure or size of the data source. You must back up the data source before executing REBUILD with the DATE NEW subcommand. We recommend that you run the utility against the copy and then replace the original file with the updated backup.

Example: Using the DATE NEW Subcommand

The following example illustrates using the DATE NEW subcommand interactively.

```
rebuild
Enter option
 1. REBUILD          (Optimize the database structure)
 2. REORG            (Alter the database structure)
 3. INDEX            (Build/modify the database index)
 4. EXTERNAL INDEX   (Build/modify an external index database)
 5. CHECK            (Check the database structure)
 6. TIMESTAMP        (Change the database timestamp)
 7. DATE NEW         (Convert old date formats to smartdate formats)
 8. MDINDEX          (Build/modify a multidimensional index)
> date new
DATE NEW
ENTER THE NAME OF THE MASTER
> employee
ENTER THE NEW NAME FOR THE MASTER
> newemp
HAVE YOU BACKED UP THE DATABASE? (YES,NO)
> yes
>   NUMBER OF ERRORS=     0
 NUMBER OF SEGMENTS=  11  ( REAL=    6  VIRTUAL=   5 )
 NUMBER OF FIELDS=    34  INDEXES=   1  FILES=     3
 TOTAL LENGTH OF ALL FIELDS=  365
HOLDING...
.
.
.
NUMBER OF SEGMENTS CHANGED=      107
```

In z/OS, the new Master File is written to ddname HOLDMAST. After the new Master File is created, you should immediately copy it to a permanent data set.

For example:

```
DYNAM COPYDD HOLDMAST(NEWEMP) MASTER(NEWEMP)
```

How DATE NEW Converts Legacy Dates

REBUILD DATE NEW subcommand overwrites the original legacy date field (an alphanumeric, integer, or packed-decimal field with date display options) with a smart date (a field in date format). When the storage size of the legacy date exceeds four bytes (the storage size of a smart date), a pad field is added to the data source following the date field:

- Formats A6YMD, A6MDY, and A6DMY are changed to formats YMD, MDY, and DMY, respectively, and have a 2-byte pad field added to the Master File.
- The storage size of integer dates (I6YMD, I6MDY, for example) is 4 bytes, so no pad field is added.

- All packed fields and A8 dates add a 4-byte pad field.

When a date is a key field (but not the last key for the segment), and it requires a pad field, the number of keys in the SEGTYPE is increased by one for each date field that requires padding.

DATE NEW only changes legacy dates to smart dates. The field format in the Master File must be one of the following (month translation edit options T and TR may be included in the format):

```
A8YYMD A8MDYY A8DMYY A6YMD A6MDY A6DMY A6YYM A6MYY A4YM A4MY

I8YYMD I8MDYY I8DMYY I6YMD I6MDY I6DMY I6YYM I6MYY I4YM I4MY

P8YYMD P8MDYY P8DMYY P6YMD P6MDY P6DMY P6YYM P6MYY P4YM P4MY
```

If you have a field that stores date values but does not have one of these formats, DATE NEW does not change it. If you have a field with one of these formats that you do not want changed, temporarily remove the date edit options from the format, run REBUILD DATE NEW, and then restore the edit options to the format.

Reference: DATE NEW Usage Notes

- The DBA password for the data source must be issued prior to issuing REBUILD.
- The original Master File cannot be encrypted.
- All files must be available locally during the REBUILD, including LOCATION files.
- The Master File cannot have GROUP fields.
- Some error numbers are available in &FOCERRNUM while all error numbers are available in &&FOCREBUILD. Test both &&FOCREBUILD and &FOCERRNUM for errors when writing procedures to rebuild your data sources.
- To avoid any potential problems, clear all LETs and JOINs before issuing REBUILD.
- DEFCENT/YRTHRESH are respected at the global, data source, and field level.
- Correct all invalid date values in the data source before executing REBUILD/DATE NEW. The utility converts all invalid dates to zero. Invalid dates used as keys may lead to duplicate keys in the data source.
- Adequate workspace must be available for the temporary REBUILD file. As a rule of thumb, have space 10 to 20% larger than the size of the existing file available.

- REBUILD/INDEX is performed automatically if an index exists.
- REBUILD/REBUILD is performed automatically after REBUILD/DATE NEW when any key is a date.
- Sort libraries and work space must be available (as with REBUILD/INDEX). The REBUILD allocates default sort work space if you have not already. DDNAMEs SORTIN and SORTOUT must be allocated prior to issuing a REBUILD.

What DATE NEW Does Not Convert

The REBUILD DATE NEW subcommand is a remediation tool for your FOCUS data sources and date fields only. It does not remediate:

- DEFINE attributes in the Master File.
- ACCEPT attributes in the Master File.
- DBA restrictions (for example, VALUE restrictions) in the Master File or central security repository (DBAFILE).
- Cross-references to other date fields in this or other Master Files.
- Any references to date fields in your FOCEXEC.

Using the New Master File Created by DATE NEW

REBUILD DATE NEW subcommand creates an updated Master File that reflects the changes made to the data source. Once the data source has been rebuilt, the original Master File can no longer be used against the data source. You must use the new Master File created by the DATE NEW subcommand.

Example: **Sample Master File: Before and After Conversion by DATE NEW**

Before Conversion	After Conversion
`FILE=filename`	`FILE=filename`
`SEGNAME=segname, SEGTYPE=S2`	`SEGNAME=segname, SEGTYPE=S3`

Before Conversion	After Conversion

`FIELD=KEY1,,USAGE=A6YMD,$`	`FIELD=KEY1,,USAGE= YMD,$`
	
	`FIELD=, ,USAGE=A2,$ PAD FIELD ADDED BY REBUILD`

`FIELD=KEY2,,USAGE=I6MDY,$`	`FIELD=KEY2,,USAGE= MDY,$`

`FIELD=FIELD3,,USAGE=A8YYMD,$`	`FIELD=FIELD3,,USAGE= YYMD,$`
	
	`FIELD=, ,USAGE=A4,$ PAD FIELD ADDED BY REBUILD`

When REBUILD DATE NEW subcommand converts this Master File:

- The SEGTYPE changes from an S2 to S3 to incorporate a 2-byte pad field.
- Format A6YMD changes to smart date format YMD.
- A 2-byte pad field with a blank field name and alias is added to the Master File.
- Format I6MDY changes to smart date format MDY (no padding needed).
- Format A8YYMD changes to smart date format YYMD.
- A 4-byte pad field with a blank field name and alias is added to the Master File.

Action Taken on a Date Field During REBUILD/DATE NEW

REBUILD/DATE NEW performs a REBUILD/REBUILD or REBUILD/INDEX automatically when a date field is a key or a date field is indexed. The following chart shows the action taken on a date field during the REBUILD/DATE NEW process.

Date Is a Key	Index	Result
No	None	NUMBER OF SEGMENTS CHANGED = *n*
No	Yes	REBUILD/INDEX on date field.
Yes	None	REBUILD/REBUILD is performed.

Date Is a Key	Index	Result
Yes	On any field	REBUILD/REBUILD is performed. REBUILD/INDEX is performed for the indexed fields.

Creating a Multi-Dimensional Index: The MDINDEX Subcommand

The MDINDEX subcommand is used to create or maintain a multi-dimensional index. For more information, see the *Describing Data* manual.

Chapter 4

Directly Editing FOCUS Databases With SCAN

SCAN is an interactive facility used for editing FOCUS and XFOCUS databases. With it, you can edit FOCUS databases using subcommands similar to those used with text editors.

Unless otherwise noted, all references to FOCUS databases also apply to XFOCUS databases.

In this chapter:

Introduction

SCAN permits you to:

- Add records to new or existing FOCUS or XFOCUS databases.
- Change field values in FOCUS databases. With SCAN, you can change the values in KEY fields (not possible with MODIFY requests).

- Delete records from FOCUS databases.
- Search through FOCUS databases to locate instances of specified character strings or values.
- Display complete record contents showing all field values, or subsets of the fields in FOCUS databases.
- Move (relink) record segments and descendant segments from one parent record to another in FOCUS databases with parent-descendant structures.

In a typical SCAN session you identify a database and locate specific logical records of interest. Your knowledge of the database's structure and contents allows you to navigate from field to field. Within the database you can add or delete instances of data at the segment level or change data values at the field level.

Note: On databases protected with DBA passwords, SCAN is only available to those who have the proper password.

As you work in a SCAN session, your changes are accumulated in a revised version of your original database. When you decide to terminate your session, you can either save the changed version of the database and overwrite the original version with it, or keep the original version as it was when you started (if you have inadvertently changed the database).

We recommend that you copy your databases before using SCAN as an additional safety precaution; SCAN is a powerful tool for manipulating data, but keeps no log of the change transactions. Using the FOCUS Absolute File Integrity feature (SET SHADOW=ON) protects you against loss of data due to system crashes. (The SET SHADOW command is only effective if it has been issued prior to database creation. Consult the *Describing Data* manual for information about the Absolute File Integrity feature. See the *Developing Applications* manual for more information about SET parameters.)

Note: Absolute File Integrity and shadow paging are not supported for XFOCUS data sources.

SCAN vs. MODIFY, HLI, and FSCAN

FOCUS includes five facilities for maintaining the data in FOCUS databases. You should be aware of their differences:

- The SCAN facility is useful for examining the data in FOCUS databases to review or physically add, change, or delete data fields. With SCAN, an experienced user can quickly adjust database contents to correct errors or update fields. To use it effectively, however, you must know the database's contents and structure.

Caution: Because SCAN works directly on the data, there is the potential for corrupting data if you are unsure of the nature of your database. For example, if a SCAN operation such as REPLACE is issued against a database field such as SALES, without adequate selection criteria, every legitimate SALES field in the database could be overwritten by the replacement value, and all field values would have to be reentered.

- MODIFY (see *Modifying Data Sources With MODIFY* on page 17) is a transaction processing environment that is used for maintaining FOCUS databases. MODIFY requests can be developed with elaborate match logic and data validation, as well as transaction logging. Such procedures, when fully tested, can be run by clerical personnel with no threat to database security.
- HLI (Host Language Interface) is an optional interface. It allows you to read and edit FOCUS databases from programs written in other programming languages (FORTRAN and C). HLI is similar to SCAN in function. HLI is described in the *Host Language Interface* manual.
- The FSCAN facility (see *Directly Editing FOCUS Databases With FSCAN* on page 389) is similar to the SCAN facility: You can view, add, change, or delete data in your FOCUS databases. The FSCAN facility provides full-screen capabilities such as a prefix area and a command line. It also provides confirmation screens for DELETE and QUIT operations. Unlike SCAN, the FSCAN facility displays parent instances that lack descendant instances (short path records) and verifies acceptable test values defined with ACCEPT parameters. MARK and MOVE subcommands are not supported.

Entering SCAN Mode

From within FOCUS, enter SCAN mode by typing SCAN followed by FILE and the name of the FOCUS database to be scanned:

```
SCAN FILE filename
```

Moving Through the Database and Locating Records

After entering SCAN, your current position is at the top of the database. FOCUS databases are not sequential databases with one data record following another; they consist of segments. Databases can have one or more segments. The segments may have multiple instances of data (a Monthly Inventory segment holding a date and a quantity might have six instances in June and twelve in December). The collected data instances for a particular set of related segments constitute a logical record in the database.

The concept of a current line pointer (common in most system editors) is replaced in SCAN by the concept of a current position in the database, which represents a set of data instances that form a connected path within the database. Instead of processing databases line-by-line, SCAN achieves a somewhat similar effect by approaching FOCUS databases in a top-down, left-to-right scanning sequence.

As we said, on entering SCAN, you are automatically positioned at the top of the database. You may move through the entire database, or specify a subset of fields to be edited (called a Show List or a subtree). Show Lists are created with the SHOW subcommand, and they contain the fields you name (plus any intermediate segments required by FOCUS to navigate from one specified field to another). An important concept when specifying Show Lists is that the data in the selected records must meet all of the criteria specified in the SHOW subcommand.

What You See in SCAN Display Lines

When you display the contents of logical records in SCAN, each data field is identified on the screen by either its alias or the field name, whichever is shorter (and non-blank). Given the following Master File, the SCAN operation proceeds as shown below.

```
FILENAME=CAR,SUFFIX=FOC
SEGNAME=ORIGIN,SEGTYPE=S1
  FIELDNAME=COUNTRY,COUNTRY,A10,FIELDTYPE=I,$
SEGNAME=COMP,SEGTYPE=S1,PARENT=ORIGIN
  FIELDNAME=CAR,CARS,A16,$
SEGNAME=CARREC,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=MODEL,MODEL,A24,$
SEGNAME=BODY,SEGTYPE=S1,PARENT=CARREC
  FIELDNAME=BODYTYPE,TYPE,A12,$
  FIELDNAME=SEATS,SEAT,I3,$
  FIELDNAME=DEALER_COST,DCOST,D7,$
  FIELDNAME=RETAIL_COST,RCOST,D7,$
  FIELDNAME=SALES,UNITS,I6,$
SEGNAME=SPECS,SEGTYPE=U,PARENT=BODY
  FIELDNAME=LENGTH,LEN,D5,$
  FIELDNAME=WIDTH,WIDTH,D5,$
  FIELDNAME=HEIGHT,HEIGHT,D5,$
  FIELDNAME=WEIGHT,WEIGHT,D6,$
  FIELDNAME=WHEELBASE,BASE,D6.1,$
  FIELDNAME=FUEL_CAP,FUEL,D6.1,$
  FIELDNAME=BHP,POWER,D6,$
  FIELDNAME=RPM,RPM,I5,$
  FIELDNAME=MPG,MILES,D6,$
  FIELDNAME=ACCEL,SECONDS,D6,$
SEGNAME=WARENT,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=WARRANTY,WARR,A40,$
SEGNAME=EQUIP,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=STANDARD,EQUIP,A40,$

 scan file car
  SCAN:
 next
 COUNTRY=ENGLAND  CAR=JAGUAR  MODEL=V12X15E AUTO
 TYPE=CONVERTIBLE SEAT= 4 DCOST= 7427 RCOST= 8878 UNITS= 0
 LEN= 190  WIDTH= 66  HEIGHT= 48  WEIGHT= 3435  BASE= 105.0
 FUEL= 18.0 BHP= 241 RPM= 5750 MPG= 16 ACCEL= 7
```

Note: SCAN uses ALIAS names instead of field names when aliases are shorter. Use DISPLAY (or CRTFORM) to display complete field names. Fields WARRANTY and STANDARD are not shown, because they do not lie on the path.

Identifying Data Fields in Scan

Some SCAN subcommands require that you specify particular data fields for the operation. LOCATE, for example, requires that you supply the data value for the target field. Within SCAN you can identify a data field in one of three ways:

- By its full field name as it appears in the Master File.
- By its alias.

- By the shortest unique truncation of either the field name or the alias.

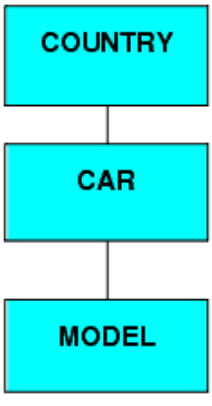
Ways to Move Through Databases

In SCAN sessions you can move from one segment instance directly to the next, jump from a parent segment instance to the first descendant field, or jump directly to a specific record of interest based on selection criteria specified in your request (for a description of these techniques, see *Subcommand Summary* on page 363).

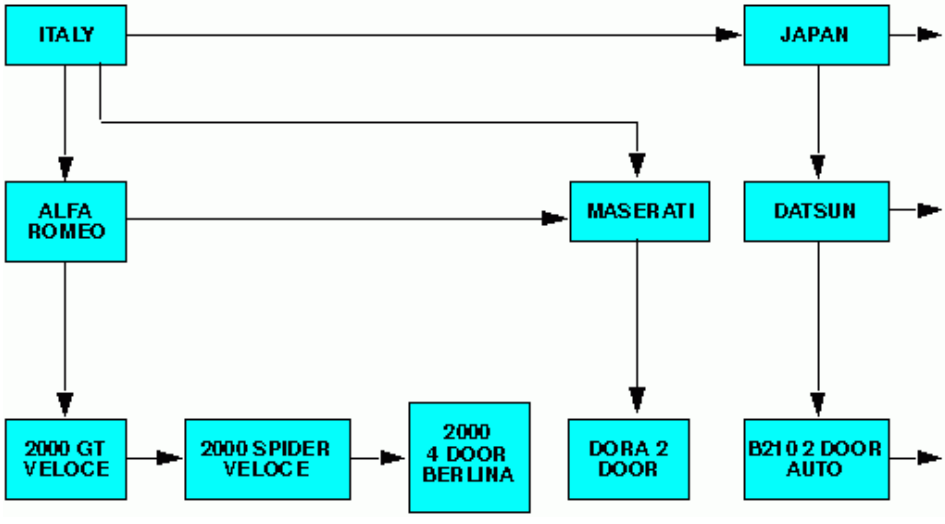
The examples in this section use the CAR database, mentioned in *What You See in SCAN Display Lines* on page 352. Enter SCAN, and then the subcommand:

```
SHOW COUNTRY CAR MODEL
```

This restricts the Show List to the first three segments of the database, as shown by this diagram:



The following schematic diagram shows how the data used in the examples is placed within the FOCUS structure:



There are six subcommands you may use to change the current position:

- *TOP* on page 355
- *LOCATE* on page 355
- *TLOCATE* on page 356
- *NEXT* on page 356
- *JUMP* on page 357
- *UP* on page 357

TOP

TOP moves the current position to the top of the database.

LOCATE

LOCATE moves the current position to the next record that fulfills certain conditions. Often, you use LOCATE to find a record with a certain value. For example, if your current position is near the top of the database and you enter the subcommand

```
LOCATE CAR=MASERATI
```

the following record appears:

```
COUNTRY = ITALY  CAR= MASERATI
                 MODEL = DORA 2 DOOR
```

If you enter this subcommand again, SCAN searches for the next MASERATI record. Since there is only one MASERATI record, it moves the current position to the end of the database.

TLOCATE

TLOCATE moves the current position to the first record in the database that fulfills certain conditions. Often, you use TLOCATE to find a record with a certain value. For example, if you enter the subcommand

```
TLOCATE CAR=ALFA ROMEO
```

the following record appears regardless of where the current position was in the database:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

NEXT

The NEXT subcommand advances the current position to the next record. That is, it advances the current position one segment instance in the lowest segment in the Show List.

Suppose you entered SCAN to edit the CAR database and displayed the first record belonging to Italy by entering:

```
TLOCATE COUNTRY=ITALY
```

SCAN displays the following record:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

You then enter the subcommand NEXT:

```
NEXT
```

The lowest segment in this example is the MODEL segment. The MODEL instance in the record (2000 GT VELOCE) is the first of three instances descended from the car ALFA ROMEO. The NEXT subcommand moves the current position to the next instance in this chain, displaying the record:

```
ITALY  ALFA ROMEO  2000 SPIDER VELOCE
```

If you enter the NEXT subcommand again, SCAN displays:

```
ITALY  ALFA ROMEO  2000 4 DOOR BERLINA
```

Now you are at the end of the MODEL under the instance ALFA ROMEO. If you enter the NEXT subcommand again, it moves the current position to the first MODEL chain under the next instance in the segment CAR. The next CAR instance is MASERATI. The record displayed is:

```
ITALY  MASERATI  DORA 2 DOOR
```

MASERATI has only one child instance, and it is the last car under the instance ITALY. If you enter the NEXT subcommand again, it moves the current position to the first MODEL chain under the next instance in the segment COUNTRY. The record displayed is:

```
JAPAN  DATSUN  B210 2 DOOR AUTO
```

JUMP

The JUMP subcommand moves the current position to the next segment instance in the segment you specify. The segment must have at least one field specified in the Show List.

Move the current position to the first record in the ITALY chain by entering:

```
TLOCATE COUNTRY=ITALY
```

This displays the record:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

Move the current position to the next car in the ITALY chain by entering:

```
JUMP CAR
```

Note: CAR is a field and not a segment name.

The following record appears:

```
ITALY  MASERATI  DORA 2 DOOR
```

Now return to the first record in the ITALY chain:

```
TLOCATE COUNTRY=ITALY
```

Jump to the next country in the database by entering:

```
JUMP COUNTRY
```

The following record appears:

```
JAPAN  DATSUN  B210 2 DOOR AUTO
```

UP

The UP subcommand moves the current position to the first instance in the lowest segment in the Show List descended from the segment that you specify.

Move the current position to the model 2000 SPIDER VELOCE:

```
TLOCATE MODEL=2000 SPIDER VELOCE
```

This displays the following record:

```
ITALY  ALFA ROMEO  SPIDER VELOCE
```

Move the current position to the first ALFA ROMEO model by entering:

```
UP CAR
```

The following records appears:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

Move the current position to the Maserati car:

```
LOCATE CAR=MASERATI
```

Move the current position to the first car in the ITALY chain by entering:

```
UP COUNTRY
```

The following record appears:

```
ITALY  ALFA ROMEO  2000 GT VELOCE
```

Displaying Field Names and Field Contents

To view up to 64 fields, specify the SHOW subcommand. The SHOW subcommand does not list records lacking instances (short-path records).

To review field contents, use either the DISPLAY or TYPE subcommand.

TYPE Subcommand

At any point in a SCAN session, you may use the TYPE subcommand to display field names in a segment path (or those named in the SHOW subcommand, if one is in effect) and their contents for the current logical record (and/or several consecutive records).

DISPLAY Subcommand

DISPLAY produces a vertical list showing the full field names followed by the data values for the current logical record. DISPLAY allows you to select the fields to be displayed, and may include fields residing in segments picked up for the subtree but not actually named in the SHOW subcommand. This displays only the fields named in the SHOW subcommand if one is in effect.

Suppressing the Display

When moving through a database in SCAN with NEXT, JUMP, LOCATE, or TLOCATE, you automatically get a display of the contents of the next record unless you suppress the display. You do this by putting a period after the move keyword. Therefore,

```
NEXT.
```

retrieves, but does not display, the next record.

It is usually preferable to suppress the displays when performing global operations that affect many records.

Show Lists and Short-Path Records

If some segments lack data, it means that some logical records have missing segment instances. FOCUS discards short-path records when constructing the Show List.

Consider a subset of the CAR database. The subset has three segments with one field per segment (COUNTRY, CAR, MODEL). If you name all three fields in a SHOW subcommand, logical records that lack data in any of the specified fields are not selected for the subtree (they are short-path records).

The following example illustrates this. To run this example, enter the following commands as shown below. What you enter is in lowercase; computer responses are in uppercase.

```
scan file car
 SCAN:
show country car
locate country=france
 COUNTRY=FRANCE CAR=PEUGEOT
input car=renault
 SCAN:
type
 COUNTRY=FRANCE CAR=RENAULT
```

The example is as follows. The CAR database contains this data:

Country	Car	Model
.		
.		
.		
France	Peugeot	504 4 DOOR

Country	Car	Model
France	Renault	
Italy	Alfa Romeo	2000 4 Door Berliner

Note that the French car Renault has no instances in the MODEL segment. A SCAN operation that names all three segments drops the logical record for Renault because Renault is missing instances in the MODEL segment, as follows.

```
show country car model
type 6
    COUNTRY=ENGLAND   CAR=JAGUAR       MODEL=V12XKE AUTO
    COUNTRY=ENGLAND   CAR=JAGUAR       MODEL=XJ12L AUTO
    COUNTRY=ENGLAND   CAR=JENSEN       MODEL=INTERCEPTOR III
    COUNTRY=ENGLAND   CAR=TRIUMPH      MODEL=TR7
    COUNTRY=FRANCE    CAR=PEUGEOT      MODEL=504 4 DOOR
    COUNTRY=ITALY     CAR=ALFA ROMEO   MODEL=2000 4 DOOR BERLINER
```

Note: In all of the examples in this section, user input is shown in lowercase; the FOCUS response is in uppercase.

To locate short-path records that will be dropped from a Show List, make a test pass through the database at the short-path level to see what is there before issuing the Show List for the edit operation. (This is highly recommended when adding new records to a database.) Thus, for the simple previous example, if you start by making a pass through the database selecting all records containing values for COUNTRY and CAR, you will find the Renault car.

```
show country car
type 6
    COUNTRY=ENGLAND  CAR=JAGUAR
    COUNTRY=ENGLAND  CAR=JENSEN
    COUNTRY=ENGLAND  CAR=TRIUMPH
    COUNTRY=FRANCE   CAR=PEUGEOT
    COUNTRY=FRANCE   CAR=RENAULT
    COUNTRY=ITALY    CAR=ALFA ROMEO
```

On the next pass, you add the MODEL segment and note that Renault disappears (due to the short-path). Knowing this, you refrain from adding a potential duplicate record for France and make a mental note to make another pass to update the short-path record with data for the MODEL segment.

Adding Segment Instances

The INPUT subcommand is used to add new segment instances to the database. New segment instances are inserted into the database in the correct sort sequence, as long as you have avoided adding duplicate instances to existing segments. Duplicate instances may not be found if they lack field values (short-path records). See *INPUT Command* on page 372 for a description of the syntax and an example of its use.

Moving Segment Instances

Use the MOVE subcommand to move a segment instance and all of its descendants from one parent segment to another. The operation requires several steps:

1. Locate the record to be moved and mark it with the MARK statement (see *MARK Command* on page 375).
2. Move the current position to the new parent record (see *LOCATE Command* on page 373 and *TLOCATE Command* on page 383).
3. Issue the MOVE subcommand indicating the field name that identifies the segment instance to be moved.

MOVE Command on page 376 describes how the segment is integrated into the database structure.

Changing Field Contents

CHANGE and REPLACE alter the contents of data fields.

Use CHANGE to substitute one character string for another, and REPLACE to substitute a new value for a field. CHANGE is issued to change alphanumeric strings within fields. REPLACE is used with either alphanumeric or numeric fields to replace the entire contents of the field(s).

Both operations can be applied to one or more instances from the current position to the end of the database. To change all instances in the database, use TLOCATE to find the first record before entering the CHANGE or REPLACE subcommand.

See *CHANGE Command* on page 366 and *REPLACE Command* on page 378 for additional information about CHANGE and REPLACE.

Deleting Fields and Segments

DELETE removes one or more instances of data in one or more segments containing the named field (and all descendant segment instances).

Saving Changes Made in SCAN Sessions

The SAVE subcommand writes all pending changes to the FOCUS database and leaves you in SCAN mode. Most installations recommend that SAVE operations be performed periodically to protect against accidental loss of update results due to communications failure or other processing interruptions.

Ending the Session

When ending the SCAN session, you can exit with or without saving your changes.

Exiting and Saving the Changes

To end the SCAN session, write the changes to the FOCUS database, and return to the FOCUS command level; use either the FILE subcommand or its synonym, END.

Exiting Without Saving the Changes

To leave SCAN and return to FOCUS without writing pending changes to the FOCUS database, use the QUIT subcommand.

Caution: The use of this subcommand does not guarantee that all changes to the database will be ignored. During SCAN execution, large buffer areas hold database records. Depending on the operating system in use and the size of these buffer areas, it is possible that a large SCAN change file could threaten the capacity of the temporary buffer storage, in which case the operating system might write the pending changes to the database to clear the buffer. This would update your database.

Auxiliary SCAN Functions

SCAN provides two convenience features: the first displays or executes a previous command; the second substitutes a one-character value for a complete SCAN subcommand.

Displaying a Previous SCAN Subcommand

To display the last subcommand issued, use the ? subcommand.

To re-execute the previous subcommand, use the AGAIN subcommand. This is particularly useful when finding multiple instances of a field value with LOCATE.

Preset X or Y to Execute a SCAN Subcommand

To set X (or Y) equal to another SCAN subcommand, type the syntax

```
{X|Y} subcommand
```

where:

subcommand

Is a SCAN subcommand.

This gives you an alias for a long, frequently-used subcommand. For example, to substitute Y for a DISPLAY subcommand showing the first and last names of the employee at the current position in the database, type:

```
Y DISPLAY FN LN
```

The next time you type Y and press the Enter key, this DISPLAY subcommand is issued.

Subcommand Summary

SCAN subcommands can be entered as unique truncations or in full. In the summary below, the capital letters represent the shortest unique truncations.

A list of descriptions of these subcommands, with additional information and examples, begins with *AGAIN Command* on page 364.

Subcommand	Function
Again	Repeat the last subcommand.
BAck	Go back to a previously marked logical record (see MArk, below).
CHAnge	Change a character string.
CRTform	Display a list of fields on a CRTFORM.
DElete	Delete one or more instances of the segment containing the named field (and all descendant segments).
DIsplay	Display the data values for the fields specified.
End	Terminate the SCAN session and write the changes to the database.
File	Terminate the SCAN session and write the changes to the database.
Input	Enter a new record.
Jump	Jump to the next or nth occurrence of field.

Subcommand	Function
`Locate`	Search for records that match the selection criteria.
`MArk`	Mark a record so that you can return to it later in the SCAN session.
`MOve`	Relink the segment to another parent.
`Next`	Move *n* records ahead.
`Quit`	End the session and drop the pending changes.
`Replace`	Replace a field value in one or more instances.
`SAve`	Save all pending changes and continue.
`SHow`	Select a subset of the fields in the database (a logical view—Show List).
`TLocate`	Go to top of database, then locate record(s) meeting the selection criteria.
`TOp`	Reset current position at first logical record in the database.
`TYpe`	Type record(s).
`UP`	Move current position to parent segment's first descendant.
`X`	Used for command substitution.
`Y`	Same as X above.
`?`	Print the previous subcommand.

AGAIN Command

The AGAIN command tells the system to repeat the previous valid command.

This is particularly useful after LOCATE, as it continues the search for the next instance of the target value.

Syntax: How to Use the AGAIN Command

```
Again
```

Example: Using the AGAIN Command

```
show emp_id last_name salary dpt
locate dpt=mis
 EID=112847612 LN=SMITH  DPT=MIS   SAL= 13200.00
again
 EID=117593129 LN=JONES  DPT=MIS   SAL= 18480.00
```

LOCATE retrieves the first record following the current position that matches the test condition. AGAIN repeats the process, as if the LOCATE statement had been retyped, and the next record that meets the test condition is displayed.

The fields displayed above are those named in the previous SHOW subcommand. The DPT (Department) field is available in the Show List because it resides in the same segment as the EMP_ID and LAST_NAME fields.

Reference: Commands Similar to Again

Within SCAN, entering a question mark (?) causes a display of the last subcommand to be executed. If you wish to execute it again, reenter the command or use AGAIN.

BACK Command

The BACK subcommand works in conjunction with the previous MARK subcommand (only one MARK is in effect at a time). When BACK is issued, control returns to the previous marked record (see MARK subcommand).

Syntax: How to Use the BACK Command

```
BAck
```

Example: Using the BACK Command

```
show emp_id last_name first_name salary
next
 EID=071382660 LN=STEVENS FN=ALFRED  SAL= 11000.00
jump emp_id 2
 EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
mark
next 2
 EID=119265415 LN=SMITH   FN=RICHARD SAL= 9500.00
back
 EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
```

Reference: Commands Similar to BACK

None.

CHANGE Command

CHANGE is used to replace specified alphanumeric character strings with new strings in data fields. Changes may be made sequentially to every record in the database, or to all records that match a LOCATE criteria.

Note: CHANGE cannot be used on numeric fields with formats I, P, F, and D.

Syntax: How to Use the CHANGE Command

```
CHAnge field=/oldstring/newstring/,$ [*|n]
```

A period (.), colon (:), or slash (/) may be used as the string delimiter and must be the first character after the equal sign (=). The same character must then be used to terminate the old and new strings.

The replication factor *n* (where *n* is number of strings to be replaced) has a default value of 1. When more than one string is to be changed, indicate the replication factor as a single digit following the line terminator characters ,$. To replace all instances of the string in the remainder of the database, use the asterisk (*). To replace all instances of the string in the database, issue TOP before the CHANGE. This resets the current position at the first logical record.

Using the CHANGE Command

This section will show how to use the CHANGE command.

Example: Single-Field Change With the CHANGE Command

To change a single field, first locate it then make the change.

```
show emp_id last_name first_name
tlocate ln=stevens
 EID=071382660 LN=STEVENS  FN=ALFRED
change ln=/stevens/stephens/,$
 EID=071382660 LN=STEPHENS FN=ALFRED
```

Example: Sequential Changes With the CHANGE Command

To change all occurrences of the old string to the new string throughout the database starting at the current position, use the replication factor, *.

```
show last_name department salary
locate dpt=mis
 LN=SMITH     DPT=MIS  SAL= 13200.00
change dpt=/mis/mis dept/,$ *
 LN=SMITH     DPT=MIS DEPT  SAL= 13200.00
 LN=JONES     DPT=MIS DEPT  SAL= 18480.00
 LN=JONES     DPT=MIS DEPT  SAL= 17750.00
 LN=MCCOY     DPT=MIS DEPT  SAL= 18480.00
 LN=BLACKWOOD DPT=MIS DEPT  SAL= 21780.00
 LN=GREENSPAN DPT=MIS DEPT  SAL= 9000.00
 LN=GREENSPAN DPT=MIS DEPT  SAL= 8650.00
 LN=CROSS     DPT=MIS DEPT  SAL= 27062.00
 LN=CROSS     DPT=MIS DEPT  SAL= 25775.00
 VALUES REPLACED= 6
 EOF:
```

The VALUE REPLACED parameter displayed at the bottom of the report shows how many segment instances were changed, not how many lines SCAN displays after the change.

Example: Match Logic Changes With the CHANGE Command

The current position is reached through a LOCATE (or TLOCATE) subcommand, and the conditions of the LOCATE are retained and applied in selecting records to be changed.

```
tlocate dpt=mis dept, sal lt 15000
 LN=SMITH      DPT=MIS DEPT  SAL=13200.00
change dpt=/mis dept/mis/ ,$ *
 LN=SMITH      DPT=MIS       SAL= 13200.00
 LN=GREENSPAN  DPT=MIS       SAL= 9000.00
 LN=GREENSPAN  DPT=MIS       SAL= 8650.00
 VALUES REPLACED= 2
 EOF:
```

Here SCAN changes only two segment instances rather than the six instances in the previous example, but three are shown because there are two child segments for the GREENSPAN record.

Note:

- If CHANGE is immediately preceded by LOCATE or TLOCATE, only the instances that satisfy the LOCATE conditions are changed.
- If no record selection criteria is included, the CHANGE action will change subsequent instances. Changed field instances may include descendant instances not represented in the Show List.

Reference: Commands Similar to CHANGE

REPLACE is used to replace the entire contents of numeric or alphanumeric fields.

CRTFORM Command

The CRTFORM subcommand formats the display of selected data fields. Enter the field names separated by blanks. (The selection begins at the current position.) The display aligns two fields per line where possible.

Use the TYPE subcommand to display the results of a CRTFORM subcommand.

Syntax: How to Use the CRTFORM Command

```
CRTform *  {*|fieldname [*]...fieldname}
```

You can enter the full field names, aliases, or the shortest unique truncations of either. To display all fields between two named fields, place an asterisk in the list of field names. To simply display all fields, use an asterisk in place of the field names.

Using the CRTFORM Command

This section shows how to use the CRTFORM command.

Example: Specifying Individual Fields With CRTFORM

```
crtform eid ln fn sal
type

EMP_ID          =071382660      LAST_NAME     =STEVENS
FIRST_NAME      =ALFRED         SALARY        = 11000.00
```

Example: Specifying All Fields Between Two Named Fields With CRTFORM

```
crtform eid * salary
type

EMP_ID           =071382660     LAST_NAME     = STEVENS
FIRST_NAME       =ALFRED        HIRE_DATE     = 800602
DEPARTMENT       =PRODUCTION    CURR_SAL      = 11000.00
CURR_JOBCODE     =A07           ED_HRS        = 25.00
BANK_NAME        =              BANK_CODE     =
BANK_ACCT        =              EFFECT_DATE   = 0
DAT_INC          =820101        PCT_INC       = .10
SALARY           =11000.00
```

Reference: Commands Similar to CRTFORM

None.

DELETE Command

The segment containing the field name is deleted and all of its descendant segments are deleted. Any references to indexed fields are removed from their associated indexes.

Note:

- If DELETE is immediately preceded by a LOCATE subcommand, then only instances that satisfy the LOCATE conditions are deleted.
- If no record selection criteria is included, the delete action will remove subsequent instances. Deleted field instances may include descendant segments that are not represented in the Show List.

None of the changes made during a SCAN session take effect until you save them. When you do write them to the database using SAVE or FILE (see descriptions of these subcommands on the following pages), they become permanent; thus you should closely monitor the effect of your changes as you work in SCAN. If you make a mistake, it is important to QUIT immediately to avoid any permanent damage.

Syntax: How to Use the DELETE Command

```
DElete fieldname [factor]
```

where:

factor

Is one of the following:

1 is the default value.

* deletes all instances of the field.

n is the number of data instances to be deleted. When more than one instance is to be deleted, indicate the replication factor as a numeric value following the line terminator characters ,$.

Example: Using DELETE

```
show emp_id last_name salary jobcode
next
 EID=071382660 LN=STEVENS  SAL=  11000.00 JBC=A07
delete jobcode 6
 SEGMENTS DELETED=  6
```

The next six instances of JOBCODE are removed.

Reference: Commands Similar to DELETE

None.

DISPLAY Command

This subcommand displays the values of the named fields in a neat vertical list, whether the field is in the SHOW list or not. It is useful to view the values of fields not specified in a SHOW list. (TYPE presents only the fields named in the SHOW command.) It is convenient, for example, to move through databases looking at only the values of a few key fields. Then, when you find the record you want, use DISPLAY to display all of the fields in the segment(s) contained in the Show List.

The DISPLAY subcommand does not remain in effect. It simply lists the specified values. If you need to issue it repeatedly, store it with the X or Y subcommand for subsequent execution.

Syntax: How to Use the DISPLAY Command

```
DIsplay fieldname [fieldname...fieldname]
```

The field identifier may be the full field name, alternate alias, or shortest unique truncation of either. Separate field names from each other with spaces.

Example: Using DISPLAY

```
show last_name dat_inc
locate ln =smith
 LN=SMITH    DI=820101
display last_name first_name salary department
LAST_NAME   =SMITH
FIRST_NAME  =MARY
SALARY      = 13200.00
DEPARTMENT  =MIS
```

If the DISPLAY subcommand does not produce a list, it indicates that the fields requested must lie outside the currently retrieved segment(s) by displaying the message:

```
NO CURRENT VALUE FOR: field.
```

Reference: Commands Similar to DISPLAY

- The TYPE subcommand is also used for showing the contents of the currently active data fields. TYPE presents the data horizontally, using the shortest name or alias available in the Master File. DISPLAY presents the information vertically, showing the full field names.

- CRTFORM is used to format a screen, showing the full field names and the field contents, blocked two to a line. Use TYPE to show the contents of the CRTFORM.

END Command

Terminates the SCAN session and writes all pending modifications to the FOCUS database.

Syntax: How to Use the END Command

```
End
```

Example: Using the END Command

```
END
```

Reference: Commands Similar to END

- The FILE subcommand is a synonym for END. This also results in normal termination of the session.
- The SAVE subcommand also writes the modifications to the database, but does not terminate the SCAN session. You retain your position in the database.

FILE Command

Terminates the SCAN session and writes all pending modifications to the FOCUS database.

Syntax: How to Use the FILE Command

```
File
```

Example: Using the FILE Command

```
FILE
```

Reference: Commands Similar to FILE

- The END subcommand is a synonym for FILE. This also results in normal termination of the session.
- The SAVE subcommand writes the modifications to the database, but does not terminate the SCAN session. You retain your position in the database.

INPUT Command

The subcommand opens the database to accept one or more new segments of data. It creates a segment instance in each segment for which a field value is specified.

The new records are inserted after the record currently displayed; that is, they break the chain. However, if the segment is being maintained in some sort sequence, a check is subsequently performed and the new records inserted in their proper positions.

Syntax: How to Use the INPUT Command

```
Input [field=value,...[,$]]
```

The input records are defined as free-format, or comma-delimited. They are entered in one of two ways:

- The data may be typed on the same line as the command. It must be typed on one line. In this case, it does not have to be terminated by a comma and dollar sign (,$).
- Or if the subcommand is issued on a line by itself, then the new record may be typed on several lines, but it *must* be terminated by a comma and dollar sign (,$).

Example: Using the INPUT Command

```
show emp_id last_name salary jobcode
tlocate ln=jones
 EID=117593129 LN=JONES SAL= 18480.00 JBC=B03
input salary=19000.00, jobcode=b04
 SCAN:
type
 EID=117593129 LN=JONES SAL= 19000.00 JBC=B04
```

Caution: SCAN rejects records that have key field values that already exist in the database (duplicate keys). In this example, if you type the following, you get a warning.

```
input eid=117593129, salary=19000.00, jobcode=604
 DATA KEYS ARE ALREADY IN FILE
 SCAN:
```

Such warnings are only provided for *key* fields, however, and inadvertently creating a duplicate instance of a segment can have unexpected consequences, particularly if one of the records is a short-path record. Subsequently, you may see different versions depending on the fields you name in your SHOW command.

Reference: Commands Similar to INPUT

None.

JUMP Command

Starting from the field in the current record, JUMP moves immediately to the next occurrence of the same field. This skips over any intervening records and is a quick way to traverse a database. Specify *n* to jump *n* occurrences.

If JUMP encounters no additional field occurrences for the same parent record, it stops at the last record in the current chain and displays the END-OF-CHAIN message. It does not move to the start of the next chain.

Syntax: How to Use the JUMP Command

```
Jump fieldname [n]
```

Example: Using the JUMP Command

```
show emp_id last_name first_name salary
type 7
 EID=071382660 LN=STEVENS FN=ALFRED  SAL= 11000.00
 EID=071382660 LN=STEVENS FN=ALFRED  SAL= 10000.00
 EID=112847612 LN=SMITH   FN=MARY    SAL= 13200.00
 EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
 EID=117593129 LN=JONES   FN=DIANE   SAL= 17750.00
 EID=119265415 LN=SMITH   FN=RICHARD SAL= 9500.00
 EID=818692173 LN=CROSS   FN=BARBARA SAL= 25775.00
 EID=119265415 LN=SMITH   FN=RICHARD SAL= 9050.00

top
 TOF:
next
 EID=071382660 LN=STEVENS FN=ALFRED  SAL= 11000.00
jump emp_id 2
 EID=117593129 LN=JONES   FN=DIANE   SAL= 18480.00
```

Reference: Commands Similar to JUMP

The NEXT subcommand is used to advance to the next logical record.

LOCATE Command

Starting at the current position, initiates a search for record(s) meeting the test condition(s). When an acceptable record is found, it is displayed. If the end of the database is encountered during the search, the message EOF: is displayed.

Syntax: How to Use the LOCATE Command

```
Locate field rel value [[AND|,]field rel value [,$] [*|n]]
```

where:

field

Is the field name of the target(s).

rel

Is one of the following test relations:

Relation	Meaning
EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS	Contains
OMITS	Omits

value

Is the object of the comparison.

n

Is the number of occurrences which may exist.

The comma-dollar sign (,$) terminator symbol is not required if only one record is sought (the default). It is required if you provide a replication factor (*n*) larger than 1. If the replication factor is set to *, then all records meeting the test conditions are displayed (from the current position to the end of the database).

When using more than one test relation, separate them by either commas or the word AND, as

```
locate field rel value, field rel value
```

or:

```
locate field rel value AND field rel value
```

If you supply a list of values with an EQ test, separate the values with the word OR:

```
locate field EQ value OR value OR value
```

Example: Using the LOCATE Command

```
show emp_id last_name first_name salary
locate dpt=mis
 EID=112847612 LN=SMITH  SAL=  13200.00 JBC=B14
```

Reference: Commands Similar to LOCATE

TLOCATE has exactly the same function, but effectively adds the TOP function and begins the search at the top of the database.

MARK Command

The MARK subcommand identifies a logical record so that you can return to it when you issue the MOVE or BACK subcommand. Only one record can be marked at a time. MARK is used to identify data to be moved to a new location in the database, and to return to a record with the BACK command.

Syntax: How to Use the MARK Command

```
MArk
```

Example: Using the MARK Command

```
show emp_id last_name first_name salary
next
 EID=071382660 LN=STEVENS  FN=ALFRED  SAL= 11000.00
jump emp_id 2
 EID=117593129 LN=JONES    FN=DIANE   SAL= 18480.00
mark
next 2
 EID=119265415 LN=SMITH    FN=RICHARD SAL= 9500.00
back
 EID=117593129 LN=JONES    FN=DIANE   SAL= 18480.00
```

Reference: Commands Similar to MARK

None.

MOVE Command

The MOVE subcommand moves segment instances and all of their descendant segments from one parent segment to another.

Identify the record instance of the segment to be moved with the MARK subcommand. Then locate the new position for the marked segment instance in any manner (LOCATE, NEXT, etc.). Follow with the MOVE subcommand naming the instance of the segment being moved. The moved instance and all of its descendants are made descendants of the parent at the current position. If the SEGTYPE is not S or SH, then the segment will be inserted after the record currently shown. If the SEGTYPE is S or SH (sorted, sorted high-to-low), the segments will be located in the proper sort sequence.

Syntax: How to Use the MOVE Command

```
MOve fieldname
```

Example: Using the MOVE Command

```
show emp_id last_name salary dat_inc
next
 EID=071382660 LN=STEVENS    DI=820101 SAL= 11000.00
mark
locate ln=greenspan
 EID=543729165 LN=GREENSPAN  DI=820611 SAL= 9000.00
move dat_inc
 EID=543729165 LN=GREENSPAN  DI=820101 SAL=11000.00
```

In the example, the date of increase (DAT_INC or DI) and salary (SAL) are taken from the marked record of Alfred Stevens and moved to Mary Greenspan's record.

Reference: Commands Similar to MOVE

None.

NEXT Command

The current position is advanced *nn* records and the new position is displayed (where nn is the number of records from 1 to 99). If the end of the database is reached during the movement to the new current position, the message EOF: is displayed.

Syntax: How to Use the NEXT Command

```
Next [nn]
```

The default is one record.

Example: Using the NEXT Command

```
show emp_id last_name first_name salary
type 8
 EID=071382660 LN=STEVENS  FN=ALFRED  SAL= 11000.00
 EID=071382660 LN=STEVENS  FN=ALFRED  SAL= 10000.00
 EID=112847612 LN=SMITH    FN=MARY    SAL= 13200.00
 EID=117593129 LN=JONES    FN=DIANE   SAL= 18480.00
 EID=117593129 LN=JONES    FN=DIANE   SAL= 17750.00
 EID=119265415 LN=SMITH    FN=RICHARD SAL= 9500.00
 EID=119265415 LN=SMITH    FN=RICHARD SAL= 9050.00
 EID=119329144 LN=BANNING  FN=JOHN    SAL= 29700.00

top
 TOF:
next 4
 EID=117593129 LN=JONES    FN=DIANE   SAL= 18480.00
next 2
 EID=119265415 LN=SMITH    FN=RICHARD SAL= 9500.00
```

NEXT 4 advances the current position to the fourth logical record and displays the field values at that position. The subsequent NEXT 2 moves the current position forward two more logical records.

Reference: Commands Similar to NEXT

None.

QUIT Command

Ends the SCAN session. All pending modifications to the database (those not yet written permanently to the disk) are suppressed.

The use of this subcommand *does not guarantee* that all changes to the database will be ignored. During SCAN execution, large buffer areas hold the pending changes. Depending on the operating system and buffer sizes, a large SCAN file could threaten the buffer capacity. This forces the operating system to write your pending changes to the database to clear the buffer. This would update your database, even though you had not issued a SAVE, END or FILE subcommand.

The FOCUS Absolute File Integrity facility reduces the risk of making changes you do not want. Also, keeping your own copy of the database before you start the session gives you a recovery capability in the event you lose your way in SCAN and create a database you subsequently decide to discard.

The QUIT subcommand acts only to prevent transfer of those records in the buffer to the disk.

When a change is made to a database immediately prior to issuing QUIT, the change is usually suppressed. If SCAN activity is high between modifications to the database, however, the chance of suppressing all changes is less likely, because the buffer work areas may, of necessity, have been written to the disk to make way for more pages of database records.

Syntax: How to Use the QUIT Command

```
Quit
```

Example: Using the QUIT Command

```
QUIT
```

Reference: Commands Similar to QUIT

END and FILE both terminate the session but both write any pending changes to the database. SAVE also writes the changes to the database, but leaves you in the SCAN session.

REPLACE Command

The REPLACE command replaces the data values for the record at the current position with the data values provided. The fields replaced may reside on the same segment or different segments, but must be on the path defined by the Show List if one is in effect.

Two types of global REPLACE operations can be specified:

- **Sequential replacement:** If the current position was not reached using a prior LOCATE subcommand, the replication factor applies to this record and the next *n*-1 records retrieved.

- **Matched replacement:** If a prior LOCATE subcommand established the current position, the search criteria remains in effect and the replication factor applies to this logical record and the next *n*-1 records that also meet the search criteria.

Syntax: How to Use the REPLACE Command

```
Replace [KEY] field=value, field=value, $ [factor]
```

where:

factor

Is one of the following:

1 is the default.

* represents all fields.

nn is the number of field values that can be replaced at one time. If the replication factor *nn* is greater than 1, then all of the replaced fields must reside on the same segment.

If the field whose value is being replaced is used to keep the segment in the proper sort sequence (that is, it is a key field), then the word KEY must be placed after the command. Without this word, a message is displayed indicating that the key field cannot be replaced.

Note: The replication factor cannot be used with REPLACE KEY.

Using the REPLACE Command

This section will show how to use the REPLACE command.

Example: Replacing a Field Value With REPLACE

```
show emp_id last_name salary
tlocate eid=112847612
 EID=112847612 LN=SMITH  SAL= 13200.00
replace salary=16000.00
 EID=112847612 LN=SMITH  SAL= 16000.00
```

Example: Replacing Multiple Field Values With REPLACE

```
show emp_id last_name jobcode
next
 EID=071382660 LN=STEVENS JBC=A07
replace jobcode=B02,$ *
 EID=071382660 LN=STEVENS JBC=B02
 EID=071382660 LN=STEVENS JBC=B02
 EID=112847612 LN=SMITH   JBC=B02
 .
 .
 .
 VALUES REPLACED= 19
 EOF:
```

Example: Replacing a Key Field Value With REPLACE

```
show emp_id last_name first_name
tlocate ln=stevens
 EID=071382660  LN=STEVENS  FN=ALFRED
 replace key eid=971382660
  EID=971382660 LN=STEVENS  FN=ALFRED
  KEY VALUE RESEQUENCED...
 type *
  EID=971382660 LN=STEVENS  FN=ALFRED
 EOF:
```

Notes on replacing key fields:

- The segment is re-sequenced to preserve the correct sort order. In this case, we gave Stevens the highest employee number in the database, so the TYPE * command types one record and reaches end-of-file.
- Only one key field can be replaced at a time.
- This may result in duplicate keys in the database (you need to keep track of this).

Reference: Commands Similar to REPLACE

CHANGE command.

SAVE Command

Writes out all modifications to the FOCUS database. The SCAN session continues at the current position held before the SAVE. If the FOCUS Absolute File Integrity feature is active, this is the point at which a new checkpoint is taken.

To activate the Absolute File Integrity feature, issue the following command at the FOCUS command level before you create the database:

```
SET SHADOW=ON
```

If the SET SHADOW command is issued after the database is created, the command has no effect. See the *Describing Data* manual for information about the FOCUS Absolute File Integrity feature. See the *Developing Applications* manual for more information about the SET parameters.

Periodic use of SAVE during SCAN sessions is recommended. Otherwise, if communication lines are lost or other processing interruptions occur, the modifications made since the previous SAVE must be repeated.

Syntax: How to Use the SAVE Command

```
SAve
```

Example: Using the SAVE Command

```
SAVE
```

All modifications to the database are written to the disk, and the SCAN session continues.

Reference: Commands Similar to SAVE

Both END and FILE write your changes to the database and terminate the SCAN session. QUIT is used to delete any pending changes to the database and terminate the SCAN session.

SHOW Command

SHOW is used to create a subset of the database (called a Show List, subtree, or a logical view) for editing. It always moves the current position to the top of the database, and the logical records are only as deep as the Show List (that is, they consist of only the segments named in the SHOW subcommand, which had data in all of the specified fields plus any intermediate segments needed to connect the segments containing the named fields).

Syntax: How to Use the SHOW Command

```
SHow [fieldlist]
```

where:

fieldlist

Can be one of the following:

fieldname [*] *fieldname* * *fieldnamefieldname* *

Separate field names with blanks. Field names can be full field names, aliases, or unique truncations of either.

On entry into the SCAN environment, all of the data fields in the first physical top-to-bottom path are displayed as the default Show List. When SHOW is issued with no list of field names, the names of all of the fields in the current path are displayed.

Use an asterisk (*) between two field names to select all fields between and including them. Use an asterisk and one field name to select all field names up to and including the named field. Use one field name and an asterisk to select all field names from that field on.

Using the SHOW Command

This sections shows how to use the SHOW command.

Example: **Selecting a Logical View (a Show List)**

```
show eid last_name salary
type *
 EID=071382660 LN=STEVENS SAL= 11000.00
 EID=071382660 LN=STEVENS SAL= 10000.00
 EID=112847612 LN=SMITH   SAL= 13200.00
 EID=117593129 LN=JONES   SAL= 18480.00
 .
 .
 .
 EID=818692173 LN=CROSS   SAL= 25775.00
 EOF:
```

The Show List, or subtree, consists of all segment instances that have data for all of the fields specified (Employee Identification Number, Last Name and Salary). Records lacking instances of any of these fields (for example, short-path records) are not included in the list.

Example: **Selecting All Fields Between Two Named Fields**

```
show emp_id * bank_name
type 2
 EID=071382660 LN=STEVENS  FN=ALFRED HDT=800602
 DPT=PRODUCTION  CSAL=11000.00 CJC=A07 OJT= 25.00 BN=
 EID=112847612 LN=SMITH    FN=MARY   HDT=810701
 DPT=MIS         CSAL=13200.00 CJC=B14 OJT= 36.00 BN=
```

All fields between (and including) EMP_ID and BANK_NAME are included in the Show List. (Stevens and Smith do not have a bank for electronic transfer and, therefore, the value for BN is blank.)

Example: **Selecting All Fields**

To select all fields, use an asterisk instead of field names.

```
SHOW *
```

Note: To examine the contents of the current position in the Show List, you can use TYPE to print just the fields named in the SHOW subcommand. Use DISPLAY or CRTFORM if you wish to see the contents of other fields in the selected segments. (Use TYPE with CRTFORM to see the display.)

Subsequent navigation keywords will show the field values for the current position for each of the fields named in the SHOW subcommand.

Reference: Commands Similar to SHOW

None.

TLOCATE Command

TLOCATE is a convenience feature that combines the capabilities of the LOCATE subcommand with those of TOP. When issued, the search begins at the top of the database. This combined functionality allows you to automate processes more easily using the X and Y subcommands.

If the subcommand AGAIN is used following TLOCATE, it locates the same record rather than moving ahead to the next instance as it would with LOCATE.

Syntax: How to Use the TLOCATE Command

```
TLocate field rel value [[AND|,]field rel value [,$][*|nn]]
```

where:

field

Is the field name of the target(s).

rel

Is one of the following test relations:

Relation	Meaning
EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to

Relation	Meaning
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS	Contains
OMITS	Omits

value

Is the object of the comparison.

The comma-dollar sign (,$) terminator character is not required if only one record is sought. However, it is required if you provide a replication factor larger than one. If the replication factor is set to *, then all records meeting the test conditions are displayed from the current position to the end of the database.

When using more than one test relation, separate them either with commas or the word AND, as follows

```
locate field rel value, field rel value
```

or:

```
locate field rel value AND field rel value
```

If you supply a list of values with an EQ test, separate the values with the word OR:

```
locate field EQ value OR value OR value
```

Example: Using the TLOCATE Command

```
show last_name first_name department
tlocate dpt=production
 LN=STEVENS  FN=ALFRED DPT=PRODUCTION
next 5
 LN=IRVING   FN=JOAN   DPT=PRODUCTION
tlocate dpt=production
 LN=STEVENS  FN=ALFRED DPT=PRODUCTION
```

Reference: Commands Similar to TLOCATE

LOCATE is the same command, but without the TOP function.

TOP Command

The current position is set at the first logical record in the database. If the next subcommand is TYPE or NEXT, the first record is retrieved and displayed.

When the message EOF: appears after any subcommand, use TOP to reset the current position.

Syntax: How to Use the TOP Command

```
TOp
```

Example: Using the TOP Command

```
show emp_id last_name salary
next 30
 EOF:
top
 TOF:
next
 EID=071382660 LN=STEVENS SAL=  11000.00
```

The current position is reset to the top of the database.

Reference: Commands Similar to TOP

- SHOW also takes you to the top of the database, but its primary purpose is the selection of the logical database view that you wish to use.
- TLOCATE goes to the top of the database before starting its search for the field(s) you have specified.

TYPE Command

The TYPE command displays the values of the named fields or displays the contents of a CRTFORM.

Syntax: How to Use the TYPE Command

```
TYpe [factor]
```

where:

factor

Is one of the following: 1 is the default.

n displays the record at the current position plus the next *n*-1 records, if the replication factor is greater than 1.

* displays the message EOF: after the last record in the database is displayed. Use TOP to reset the current position to the top of the database.

Example: Using the TYPE Command

```
show emp_id last_name salary
type 6
 EID=071382660 LN=STEVENS  SAL= 11000.00
 EID=071382660 LN=STEVENS  SAL= 10000.00
 EID=112847612 LN=SMITH    SAL= 13200.00
 EID=117593129 LN=JONES    SAL= 18480.00
 EID=117593129 LN=JONES    SAL= 17750.00
 EID=119265415 LN=SMITH    SAL= 9500.00
```

The record at the current position and the next five records are displayed.

Reference: Commands Similar to TYPE

- The DISPLAY command also shows the contents of the currently active data fields, but DISPLAY shows all the named fields in a neat vertical list, whether they are in the SHOW command or not.
- CRTFORM is used to format a screen, showing the full field names and the field comments, blocked two to a line. Use TYPE to show the contents of the CRTFORM.

UP Command

The UP subcommand resets the current position to the first descendant instance under a parent instance. Hence, it moves the position to the start of the current chain.

Syntax: How to Use the UP Command

```
UP fieldname
```

where:

fieldname

Is the name of a field in a descendant segment.

Example: Using the UP Command

```
show emp_id last_name salary pay_date
next 5
 EID=071382660 LN=STEVENS  SAL= 10000.00 PD=820630
up pay_date
 EID=071382660 LN=STEVENS  SAL= 10000.00 PD=820528
```

The current position is reset to the first instance of PAY_DATE information for Stevens.

Reference: Commands Similar to UP

None.

X and Y Commands

The X and Y subcommands are used to store a complete SCAN subcommand for later execution by simply typing in the appropriate letter (X or Y).

To set, but not execute, a value for X or Y, type it as a first letter in front of any other subcommand. Any print suppression control, and the replication factors, are picked up from the stored subcommand.

Syntax: How to Use the X and Y Commands

```
[x|y] subcommand
```

Example: Using the X and Y Commands

```
y display emp_id last_name curr_sal pay_date gross
show emp_id pay_date
next
 EID=071382660         PD=820831
y
 EMP_ID    =071382660
 LAST_NAME =STEVENS
 CURR_SAL  =  11000.00
 PAY_DATE  =820831
 GROSS     =   916.67
```

A series of operations can be performed by repeatedly entering X and Y subcommands.

Reference: Commands Similar to X and Y

None.

? Command

The ? subcommand recalls and displays the last recognized subcommand issued in the SCAN mode.

Syntax: How to Use the ? Command

```
?
```

Example: Using the ? Command

```
Show emp_id last_name salary jobcode
locate dpt=mis
 EID=112847612 LN=SMITH  SAL= 13200.00 JBC=B14
?
 LOCATE DPT=MIS
again
 EID=117593129 LN=JONES  SAL= 18480.00 JBC=B03
```

Here the LOCATE operation returns a record. AGAIN locates the next record that meets the stated criteria.

Reference: Commands Similar to ?

None.

Chapter 5

Directly Editing FOCUS Databases With FSCAN

The full-screen FSCAN facility enables you to edit FOCUS databases directly on your terminal screen. You can use FSCAN to add, update, and delete data from FOCUS databases as if the segments in the FOCUS databases were flat files on a full-screen editor. You can type over field values, or change them by issuing commands.

In this chapter:

Introduction

FSCAN enables you to:

- Add records to new or existing FOCUS databases.
- Change field values in FOCUS databases. With FSCAN you can change the values in key fields (not possible with MODIFY requests).
- Delete records from FOCUS databases.
- Search through FOCUS databases to locate instances of specified character strings or values.

If your database is protected by shadow paging, the changes you make on FSCAN are not permanent until you issue a command to do so. You may choose to exit FSCAN without saving any of the changes.

Databases on Which FSCAN Can Operate

FSCAN can operate on databases having the following attributes:

- The databases are FOCUS databases, not databases of other types.
- The databases are individual databases, not combined structures created by the COMBINE command.
- The length of the root key field in the database does not exceed 61 bytes, and the sum of the field name length plus the field length does not exceed 73 bytes.

Also, note the following regarding databases:

- FSCAN does not accept alternate file views.
- Databases that you specify with the USE command using the READ option are write protected.
- Databases that you are viewing on a FOCUS Database Server in Simultaneous Usage mode are write protected.

Segments on Which FSCAN Can Operate

The following rules apply to the display and editing of segments in FSCAN:

- FSCAN does not display a segment containing a key field longer than 61 bytes and the sum of the field name length plus the field length does not exceed 73 bytes, nor does it display the descendants of that segment.
- When you input a new segment instance, the instance must have a key unique to its group. (In the root segment, this means all the instances in the segment; in a descendant segment, this means all the instances that share a parent instance). If you try to input an instance with a duplicate key, FSCAN will generate an error message.
- If you change a key field value of an instance, the new instance key (the combination of all key field values in the instance) must be unique to the group. If you try to change the key to a duplicate, FSCAN will generate an error message.
- If you use FSCAN on segments already containing duplicate keys, the results are unpredictable. If the root segment has duplicate keys, an attempt to display a screen with these duplicates results in FSCAN terminating in an error. If a descendant segment has duplicate keys, an FSCAN error is displayed and you are positioned at the parent segment.

- When a segment is type S0 or blank, no one field is designated as the key field. FSCAN considers all fields in such segments to be key fields. This has two ramifications:
  - You cannot input a segment instance that is the duplicate of another in the same group.
  - You cannot update a segment instance so that it duplicates another segment instance in the same group.

Fields That FSCAN Can Display

FSCAN can display fields containing the following attributes:

- The field length does not exceed 61 bytes and the sum of the field name length plus the field length does not exceed 73 bytes.
- The fields are real database fields, not DEFINEd fields.
- FSCAN displays group fields as their individual members, not as a group.

Note: Text fields cannot be displayed in FSCAN.

Database Integrity Considerations

How FSCAN treats the changes you make to the database depends on whether the database is protected by shadow paging.

If you are using shadow paging, FSCAN writes your changes to a shadow database. If you enter the commands END, FILE, or SAVE, the changes become part of the real database. If you enter the command QQUIT or if FSCAN terminates abnormally, the changes disappear and the database is not affected.

If you are not using shadow paging, FSCAN writes your changes directly to the database. The changes remain even after you enter the QQUIT command.

FOCUS performs shadow paging using the Absolute File Integrity facility.

Note: Absolute File Integrity and shadow paging are not supported for XFOCUS data sources.

DBA Considerations

If the database is protected by the DBA security facility, then the ACCESS attribute in the Master File restricts users in the following way:

- Users with read-write access (ACCESS=RW) and write-only access (ACCESS=W) have unrestricted access to the database, with the exception of what is denied them by the RESTRICT and NAME attributes.

- Users with update-only access (ACCESS=U) can display the entire database, with the exception of what is denied them by the RESTRICT and NAME attributes. However, they cannot input or delete instances and can only update non-key fields.
- Users with read-only access (ACCESS=R) to any part of the database cannot use FSCAN on the database.

FSCAN honors DBA security restrictions on segments and fields. FSCAN does not display those segments and fields from which the user is restricted. FSCAN does not honor DBA field value restrictions and will display all field values regardless of the user.

If the user has no access to a key field in the root segment, that user is blocked from using FSCAN on the database.

If the user has no access to a segment, that segment is not listed on the menu that appears when the user enters the CHILD command.

Entering FSCAN

Enter the full-screen FSCAN facility from FOCUS with

```
FSCAN FILE filename
```

where:

filename

Is the name of the database you are editing. The database must be a FOCUS database. You may also enter FSCAN by typing:

```
FS FILE filename
```

For example, to edit the EMPLOYEE database, enter:

```
FSCAN FILE EMPLOYEE
```

Entering FSCAN With a SHOW List

By default, FSCAN makes all fields in the database available to the user. However, it is possible to restrict the fields available with the SHOW option.

Syntax: How to Enter FSCAN With a SHOW List

```
FSCAN FILE filename SHOW
[fieldname.....fieldname....|SEG.fieldname]
END
```

where:

SHOW

Indicates that specific fields will be displayed. The SHOW keyword must appear on the same line as the FSCAN command.

fieldname...

Are the fields to be displayed.

END

Is required and must be specified on a line by itself.

Example: **Entering FSCAN With a SHOW List**

For example, the commands

```
FSCAN FILE EMPLOYEE SHOW
EMP_ID LAST_NAME FIRST_NAME SEG.GROSS
END
```

would provide access to only the selected fields in the root segment and to the whole segment containing the field GROSS. The above commands would produce the following display:

```
FSCAN   FILE   EMPLOYEEFOCUS   A               CHANGES :0

         EMP_ID       LAST_NAME    FIRST_NAME
------   ---------    ---------    ----------
==       071382660    STEVENS      ALFRED
==       112847612    SMITH        MARY
==       117593129    JONES        DIANE
==       119265415    SMITH        RICHARD
==       119329144    BANNING      JOHN
==       123764317    IRVING       JOAN
==       126724188    ROMANS       ANTHONY
==       219984371    MCCOY        JOHN
==       326179357    BLACKWOOD    ROSEMARIE
==       451123478    MCKNIGHT     ROGER
==       543729165    GREEENSPAN   MARY
---------------------------INPUT-----------------------------------
==
==>

                                                            MORE=>
```

The only child segment that can be displayed is the SALINFO segment, which contains the field GROSS.

Allowing Uppercase and Lowercase Alpha Fields

By default, FSCAN translates all input and changed alpha fields to uppercase. If uppercase and lowercase input and updates are to be respected, then enter FSCAN with the LOWER keyword.

Syntax: How to Specify Case Sensitivity in FSCAN

```
FSCAN FILE filename [case]
```

where:

case

Is one of the following:

UPPER translates all input and changed alpha fields into uppercase. UPPER is the default.

LOWER preserves uppercase and lowercase input and is analogous to the CRTFORM LOWER statement in MODIFY.

MIXED is a synonym for LOWER.

Using FSCAN

When you enter FSCAN, FSCAN displays as much as it can of the root segment of the data source. For example, if you view the EMPLOYEE data source with FSCAN, using the following command

```
FSCAN FILE EMPLOYEE
```

you will see the following screen:

```
1. FSCAN FILE   EMPLOYEEFOCUS   A1              CHANGES:     0

2. EMP_ID        LAST_NAME     FIRST_NAME     HIRE_DATE      DEPARTMENT
   ------        ---------     ----------     ---------  -----------
3. == 071382660  STEVENS       ALFRED          800602         PRODUCTION
4. == 112847612  SMITH         MARY            810701         MIS
   == 117593129  JONES         DIANE           820501         MIS
   == 119265415  SMITH         RICHARD         820104         PRODUCTION
   == 119329144  BANNING       JOHN            820801         PRODUCTION
   == 123764317  IRVING        JOAN            820104         PRODUCTION
   == 126724188  ROMANS        ANTHONY         820701         PRODUCTION
   == 219984371  MCCOY         JOHN            810701         MIS
   == 326179357  BLACKWOOD     ROSEMARIE       820401         MIS
   == 451123478  MCKNIGHT      ROGER           820202         PRODUCTION
   -------------------------------INPUT-----------------------------------
5. ==
6. ==>
7.                                                                MORE=>
```

This screen displays the contents of the root segment of the EMPLOYEE database. Each record on the screen is one instance in the root segment. The numbers in the diagram refer to the notes below:

1. The header shows the name of the database and the number of changes made to the database since the last save.
2. Each field is labeled with a column heading.
3. The first record at the top of the screen is called the current instance. Many commands operate only on this record. When you first enter FSCAN, this record is the first instance in the root segment.
4. The equal signs (==) in the left margin of the screen indicate the prefix area. This is where you enter prefix area commands.

   The key field value in each record appears highlighted.
5. The last line with equal signs is called the input area and is reserved exclusively for input.
6. The arrow at the lower-left corner of the screen points to the command line. This is where you enter FSCAN commands.
7. The MORE symbol at the lower-right corner of the screen indicates that each record extends to the right of the screen.

This section discusses various functions of the FSCAN facility. For an alphabetic summary of commands, see *Syntax Summary* on page 427.

The FSCAN facility displays one segment at one time. (For the root segment, FSCAN displays all instances in the segment; for descendant segments, FSCAN displays all instances sharing the same parent instance.) Each record on the screen is one segment instance. The first instance at the top of the screen is called the current instance.

The FSCAN facility also displays segments in SINGLE mode, that is, one instance at one time. SINGLE mode is discussed in *Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands* on page 414.

Note the different types of commands:

- **Prefix area commands** are typed in the prefix area on the left of the screen display. Prefix area commands operate only on the line where they are typed.
- **Command-line commands** are typed on the command line at the bottom of the screen. Some commands operate on the entire screen, others operate only on the current instance at the top of the screen. There are two types of command-line commands:
  - **Immediate commands.** When you execute an immediate command, the database remains unchanged even if you typed changes on the screen. There are five immediate commands:

```
LEFT
RIGHT
RESET
?
QQUIT
```

- **Non-immediate commands.** When you execute a non-immediate command, any changes you type on the screen will be written to the database even if the command itself does not modify the database.

The following rules apply to commands:

- You may use unique truncations for commands. When this section specifies a command syntax, the unique truncation is shown in uppercase.
- Commands that use field names as parameters require the full field name, alias, or unique truncation.
- You may enter two commands at one time by separating the commands with a semicolon. For example, to enter the commands NEXT 5 and CHILD at one time, type:

```
NEXT 5; CHILD
```

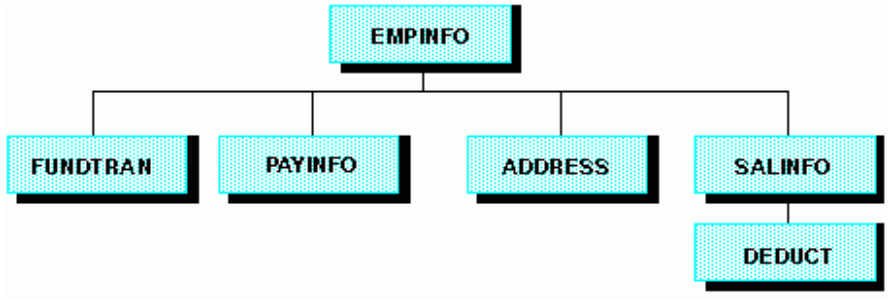
The FSCAN Facility and FOCUS Structures

This section is a brief summary of FOCUS structures and how they affect the FSCAN facility.

FOCUS databases are organized into segments which have the following properties:

- Segments consist of individual data records called segment instances, in which fields have a one-to-one correspondence with each other.
- Segments relate to each other as parents and children.
- A group of instances in a child segment describes one instance in a parent segment.
- One parent segment may have many child segments, but a child segment may have only one parent.
- A FOCUS structure has one segment from which all other segments are descended. This is called the root segment.

The diagram below represents the structure of the EMPLOYEE database:



Note the position of the segments in the structure:

- The EMPINFO segment is the root segment. All other segments are descended from it.
- EMPINFO has four children: the FUNDTRAN, PAYINFO, ADDRESS, and SALINFO segments.
- The SALINFO segment has one child, the DEDUCT segment.

The FSCAN facility displays instances in one segment at one time. When it displays the root segment (as it will when you first enter FSCAN), it displays all the instances in the segment.

The following screen illustrates how FSCAN displays the EMPINFO segment.

```
FSCAN   FILE   EMPLOYEEFOCUS    A1             CHANGES :0

        EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE   DEPARTMENT
        ------      ---------  ----------  ---------   ----------
==      071382660   STEVENS    ALFRED      800602      PRODUCTION
==      112847612   SMITH      MARY        810701      MIS
==      117593129   JONES      DIANE       820501      MIS
==      119265415   SMITH      RICHARD     820104      PRODUCTION
==      119329144   BANNING    JOHN        820801      PRODUCTION
==      123764317   IRVING     JOAN        820104      PRODUCTION
==      126724188   ROMANS     ANTHONY     820701      PRODUCTION
==      219984371   MCCOY      JOHN        810701      MIS
==      326179357   BLACKWOOD  ROSEMARIE   820401      MIS
==      451123478   MCKNIGHT   ROGER       820202      PRODUCTION
---------------------------------INPUT---------------------------------
--
==
==>
                                                       MORE=>
```

Note that the screen only displays the first five fields of the first ten instances in the segment. To view the other fields and instances, use the scrolling facilities described in *Scrolling the Screen* on page 400.

Also note that you cannot move from one segment to another by simply scrolling. To move from a parent segment to a child segment and back again, you must use the PARENT and CHILD commands discussed in *Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 411.

When FSCAN displays a child segment, it displays only those instances relating to an instance in the parent segment. You can scroll back and forth to view all the instances in the group, but you cannot scroll to view the child instances of another parent. At the top of the screen, FSCAN displays up to five keys of the parent instance, and of the parent of the parent, and so on, up to the root segment.

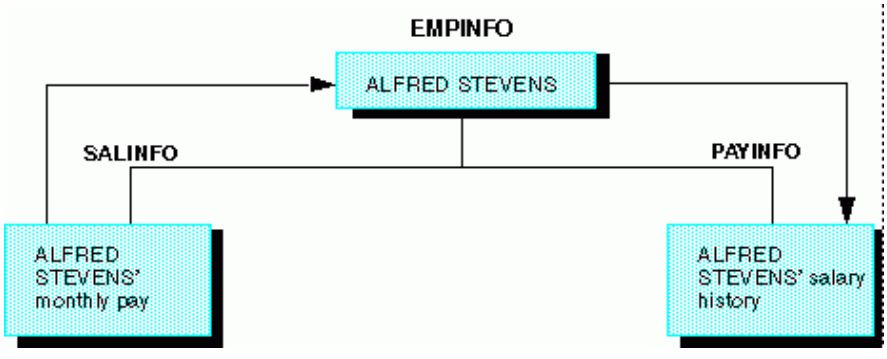
For example, the EMPINFO segment contains the ID numbers and names of employees; its child (SALINFO) contains monthly pay instances. (Each instance lists how much each employee was paid each month.) Each group of instances in SALINFO represents all the monthly pay of one employee recorded in the EMPINFO segment. When FSCAN displays the SALINFO segment, it displays one group of instances at one time.

This is how FSCAN displays the monthly pay of Alfred Stevens, who is listed in the EMPINFO segment. Note that Mr. Stevens' employee ID (the EMPINFO key field) appears at the top of the screen:

```
FSCAN  FILE  EMPLOYEEFOCUS    A1              CHANGES :0

EMP_ID : 071382660

          PAY_DATE     GROSS
          --------     -----
==        820831       916.67
==        820730       916.67
==        820630       916.67
==        820528       916.67
==        820430       916.67
==        820331       916.67
==        820226       916.67
==        820129       916.67
==        811231       833.33
--------------------------------INPUT---------------------------------
--
==


==>
```

If you are displaying one child segment and wish to display another one, you must return to the parent and request the other child segment. For example, if you are examining Alfred Stevens' monthly pay and wish to view his salary history (contained in the segment PAYINFO), return to the EMPINFO segment and request PAYINFO information for Alfred Stevens using the CHILD command described in *Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 411.

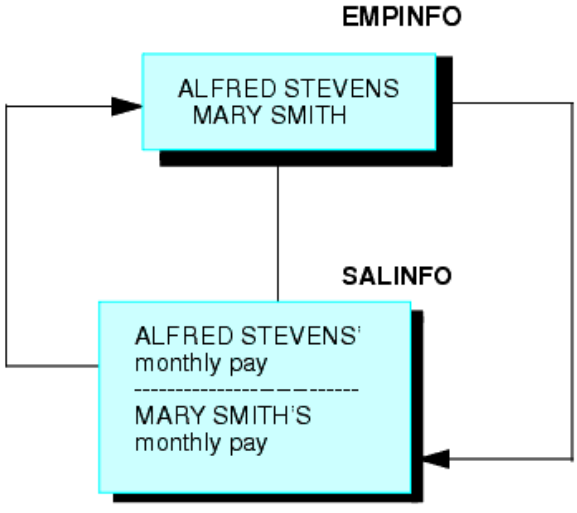
The figure below shows this path schematically. The arrows show the direction you are traveling to move from the SALINFO segment to the PAYINFO segment:



Similarly, if you are displaying one group of child instances and wish to display another group within the same segment but belonging to a different instance in the parent, you must return to the parent segment and request the child segment for the other instance.

For example, suppose you are examining Alfred Stevens' monthly pay and wish to view Mary Smith's monthly pay. You must return to the EMPINFO segment and select the SALINFO segment for Mary Smith.

The figure below shows this path schematically. The arrows show the direction you are traveling to move from Alfred Stevens' monthly pay instances to Mary Smith's monthly pay instances:



Scrolling the Screen

You may scroll the screen forward and backward, right and left.

Syntax: How to Scroll the Screen Forward

To scroll forward one screen in a segment, enter

```
FOrward
```

or press the PF8 or PF20 key. Note that the last instance on one screen becomes the first instance on the next screen.

To scroll the screen *n* lines forward, enter

```
Next n
```

or:

```
DOwn n
```

If you do not enter a number for *n*, the default is 1.

Example: **Scrolling Forward**

For example, suppose the screen displays the EMPLOYEE root segment as shown below.

```
FSCAN FILE EMPLOYEEFOCUS A1         CHANGES :0

     EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
     ------      ---------  ----------  ---------  ----------
==   071382660   STEVENS    ALFRED      800602     PRODUCTION
==   112847612   SMITH      MARY        810701     MIS
==   117593129   JONES      DIANE       820501     MIS
==   119265415   SMITH      RICHARD     820104     PRODUCTION
==   119329144   BANNING    JOHN        820801     PRODUCTION
==   123764317   IRVING     JOAN        820104     PRODUCTION
==   126724188   ROMANS     ANTHONY     820701     PRODUCTION
==   219984371   MCCOY      JOHN        810701     MIS
==   326179357   BLACKWOOD  ROSEMARIE   820401     MIS
==   451123478   MCKNIGHT   ROGER       820202     PRODUCTION
---------------------------------INPUT-------------------------------
==
==> forward

                                                              MORE=>
```

When you type the FORWARD command on the command line and press *Enter*, the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1           CHANGES :0

     EMP_ID      LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
     ------      ---------  ----------  ---------  ----------
==   451123478   MCKNIGHT   ROGER       820202     PRODUCTION
==   543729165   GREENSPAN  MARY        820401     MIS
==   818692173   CROSS      BARBARA     811102     MIS




---------------------------------INPUT-------------------------------
==
==>

                                                              MORE=>
```

Syntax: How to Scroll the Screen Backward

To scroll the screen backward, enter

```
Backward
```

or press the PF7 or PF19 key.

Syntax: How to Scroll the Screen to the Right and the Left

To scroll the screen one panel to the right, enter

```
RIght
LEft
```

or press the PF11 or PF23 key.

To scroll the screen one panel to the left, enter

```
LEft
```

or press the PF10 or PF22 key.

The commands RIGHT and LEFT are immediate commands. When you scroll right and left, FSCAN does not enter changes you typed on the screen until you press Enter after scrolling.

Example: Scrolling the Screen

For example, if you scroll the EMPLOYEE root segment display one panel to the right, the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1         CHANGES :0

    CURR_SAL         CURR_JOBCODE    ED_HRS
    --------         ------------    ------
==  11000.00         A07              25.00
==  13200.00         B14              36.00
==  18480.00         B03              50.00
==  9500.00          A01              10.00
==  29700.00         A17                .00
==  26862.00         A15              30.00
==  21120.00         B04               5.00
==  18480.00         B02                .00
==  21780.00         B04              75.00
==  16100.00         B02              50.00
----------------------------------INPUT--------------------------------
==
==>

                                                                MORE=>
```

Selecting a Specific Instance by Defining a Current Instance

This section describes how to move through the database by defining a particular instance as the current instance. The current instance is always the top instance on the screen. Certain commands only operate on the current instance.

Procedure: How to Define a Current Instance

To define an instance as the current instance, type a slash (/) in the prefix area corresponding to the instance.

You may also type a slash before or after the following prefix area commands:

- The K command (K/ or /K). After FSCAN changes the key field and displays the instance in proper sequence, it makes the instance the current instance.
- The I command (I/ or /I). After FSCAN adds a new instance to the database, it makes the instance the current instance.

Example: Defining a Current Instance: The "/" Prefix

For example, suppose you type a slash in the prefix area of John Banning's instance, as shown below:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :0

   EMP_ID     LAST_NAME   FIRST_NAME  HIRE_DATE   DEPARTMENT
   ------     ---------   ----------  ----------  ----------
== 071382660  STEVENS     ALFRED      800602      PRODUCTION
== 112847612  SMITH       MARY        810701      MIS
== 117593129  JONES       DIANE       820501      MIS
== 119265415  SMITH       RICHARD     820104      PRODUCTION
/= 119329144  BANNING     JOHN        820801      PRODUCTION
== 123764317  IRVING      JOAN        820104      PRODUCTION
== 126724188  ROMANS      ANTHONY     820701      PRODUCTION
== 219984371  MCCOY       JOHN        810701      MIS
== 326179357  BLACKWOOD   ROSEMARIE   820401      MIS
== 451123478  MCKNIGHT    ROGER       820202      PRODUCTION
---------------------------------INPUT-------------------------------
==
==>
                                                              MORE=>
```

When you press *Enter*, the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :0

   EMP_ID     LAST_NAME   FIRST_NAME  HIRE_DATE   DEPARTMENT
   ------     ---------   ----------  ---------   ----------
== 119329144  BANNING     JOHN        820801      PRODUCTION
== 123764317  IRVING      JOAN        820104      PRODUCTION
== 126724188  ROMANS      ANTHONY     820701      PRODUCTION
== 219984371  MCCOY       JOHN        810701      MIS
== 326179357  BLACKWOOD   ROSEMARIE   820401      MIS
== 451123478  MCKNIGHT    ROGER       820202      PRODUCTION
== 543729165  GREENSPAN   MARY        820401      MIS
== 818692173  CROSS       BARBARA     811102      MIS
---------------------------------INPUT-------------------------------
==
==>
                                                              MORE=
```

Syntax:

How to Define the First and Last Instances of a Segment on Display: The FIRST, LAST, and TOP Commands

FSCAN displays all instances in a segment that share a common parent instance. For the root segment, this means all the instances in the segment. To define the first instance in the group as the current instance, enter:

```
FIrst
```

If you are displaying instances in the root segment, FIRST will make the first instance in the database the current instance. If you are displaying instances in a child segment and use the FIRST command, the first child instance will become the current instance.

To define the last instance as the current instance, enter:

```
LAst
```

To select the first instance in the root segment of the database to be the current instance, enter:

```
Top
```

TOP displays the root segment, scrolled to the leftmost panel, with the first instance the current instance.

Example: **Defining the Last Instance as the Current Instance With LAST**

For example, if you enter LAST on the EMPLOYEE root segment display, the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :0

   EMP_ID    LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
   ------    ---------  ----------  ---------  ----------
== 818692173 CROSS      BARBARA     811102     MIS




--------------------------------INPUT---------------------------------
==
==>

                                                              MORE=>
```

Syntax: **How to Locate an Instance Based on Field Values: The LOCATE Command**

LOCATE searches for instances containing field values that fulfill certain conditions. For example, it can search for an instance with a LAST_NAME value of BANNING or a CURR_SAL value less than 20,000. LOCATE searches starting with the current instance.

The syntax is (entered on one line)

```
LOcate field1 rel1 value1 [OR value1a OR value1b OR ...]
[{AND|,} field2 rel2 value2 {AND|,} ...]
```

where:

fieldn ...

Is a field to be tested.

reln ...

Is one of the following condition relations:

EQ or =	Equal to

NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance with a field value that passes the test becomes the current segment.

If you supply more than one test condition in the command, FSCAN searches for the instance that fulfills all of the conditions. Separate the test conditions in the command with the word AND or with a comma (,).

OR

Enables you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in a single LOCATE command.

The LOCATE command searches starting with the first instance following the current instance. If LOCATE cannot find the instance, it displays a message and the current instance does not change.

Example: **Locating an Instance Based on Field Values**

For example, suppose the first instance in the EMPLOYEE root segment is the current instance. If you issue the command

```
LOCATE LAST_NAME EQ SMITH
```

the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1        CHANGES :0

   EMP_ID      LAST_NAME      FIRST_NAME     HIRE_DATE      DEPARTMENT
   ------      ---------      ----------     ---------       ----------
== 112847612   SMITH          MARY           810701         MIS
== 117593129   JONES          DIANE          820501         MIS
== 119265415   SMITH          RICHARD        820104         PRODUCTION
== 119329144   BANNING        JOHN           820801         PRODUCTION
== 123764317   IRVING         JOAN           820104         PRODUCTION
== 126724188   ROMANS         ANTHONY        820701         PRODUCTION
== 219984371   MCCOY          JOHN           810701         MIS
== 326179357   BLACKWOOD      ROSEMARIE      820401         MIS
== 451123478   MCKNIGHT       ROGER          820202         PRODUCTION
== 543729165   GREENSPAN      MARY           820401         MIS
-------------------------------INPUT-----------------------------------
==
==>
                                                                MORE=>
```

These are other examples of the LOCATE command:

```
LOCATE JOBCODE EQ A07 OR A17
```

This LOCATE searches for the first segment instance that has a JOBCODE value of either A07 or A17.

```
LOCATE LAST_NAME CO WOOD
```

This LOCATE searches for the first segment instance with a LAST_NAME value that contains the character string WOOD.

```
LOCATE HIRE_DATE GT 820401 AND JOBCODE IS B02 OR B03
```

This LOCATE searches for the first segment instance with both a HIRE_DATE value greater than 820401 and a JOBCODE value that is either B02 or B03.

Syntax: How to Find an Instance in a Group: The FIND Command

The FIND command works within the group of instances being displayed. In the root segment, this is all instances in the segment; in descendant segments, this is all instances sharing a common parent instance. FIND searches for instances containing field values that fulfill certain conditions. For example, it can search for an instance with a LAST_NAME value of BANNING or a CURR_SAL value less than 20,000. FIND searches starting with the current instance.

The syntax is entered on one line.

```
FIND field1 rel1 value1 [OR value1a OR value1b OR ...]
[{AND|,} field2 rel2 value2 {AND|,} ...]

[{AND|,} field2 rel2 value2 {AND|,} ...]
```

where:

fieldn ...

Is a field in the segment.

reln ...

Is one of the following condition relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance with a field value that passes the test becomes the current segment.

If you supply more than one test condition in the command, FSCAN searches for the instance that fulfills all of the conditions. Separate the test conditions in the command with the word AND or with a comma (,).

OR

Enables you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in a single FIND command.

The FIND command searches the group starting with the first instance following the current instance. To search the entire group, issue the FIRST command before issuing FIND. If FIND cannot find the instance, it displays a message and the current instance does not change.

Example: Finding an Instance in a Group

For example, suppose the first instance in the EMPLOYEE root segment is the current instance. If you issue the command

```
FIND LAST_NAME EQ SMITH
```

the following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :0

    EMP_ID     LAST_NAME     FIRST_NAME  HIRE_DATE  DEPARTMENT
    ------     ---------     ----------  ---------  ----------
==  112847612  SMITH         MARY        810701     MIS
==  117593129  JONES         DIANE       820501     MIS
==  119265415  SMITH         RICHARD     820104     PRODUCTION
==  119329144  BANNING       JOHN        820801     PRODUCTION
==  123764317  IRVING        JOAN        820104     PRODUCTION
==  126724188  ROMANS        ANTHONY     820701     PRODUCTION
==  219984371  MCCOY         JOHN        810701     MIS
==  326179357  BLACKWOOD     ROSEMARIE   820401     MIS
==  451123478  MCKNIGHT      ROGER       820202     PRODUCTION
==  543729165  GREENSPAN     MARY        820401     MIS
---------------------------------INPUT--------------------------------
==
==>
                                                          MORE=>
```

These are other examples of the FIND command:

```
FIND DEPARTMENT EQ MIS OR SALES
```

This FIND searches for the first segment instance that has a DEPARTMENT value of either MIS or SALES.

```
FIND LAST_NAME CO WOOD
```

This FIND searches for the first segment instance with a LAST_NAME value that contains the character string WOOD.

```
FIND HIRE_DATE GT 820401 AND DEPARTMENT EQ MIS OR PRODUCTION
```

This FIND searches for the first segment instance with both a HIRE_DATE value greater than 820401 and a DEPARTMENT value that is either MIS or PRODUCTION.

Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands

The CHILD, PARENT, and JUMP commands enable you to display the data in different segments of a data source.

Syntax: How to Display a Child Segment

To display instances in a child segment relating to the current instance, enter

```
CHIld
```

or press PF5 or PF17. If the segment on the screen when you enter the command has only one child segment, FSCAN shows the child segment. If the segment on the screen has more than one child segment, FSCAN displays a menu of child segments. Select a segment by entering its number. (**Note:** The menu does not display segments restricted to you as a result of DBA restrictions.)

If you already know the number of the segment on the menu, you can skip the menu by entering

```
CHIld n
```

where:

n

is the number of the segment on the menu.

You can display the child instances of any instance on the screen by typing C in the prefix area next to the instance. You can skip the menu by typing C followed by the number of the segment on the menu.

Example: Displaying a Child Segment

For example, suppose you are displaying the root segment of the EMPLOYEE database and you want to see the monthly pay of Mary Smith. Monthly pay is contained in the segment SALINFO, a child of the root segment. First, make Mary Smith's instance the current instance. Then, enter the command:

```
CHILD
```

The following menu appears:

```
FSCAN FILE EMPLOYEEFOCUS A1              CHANGES :0

 Please enter the number of the child segment you want

 1)FUNDTRAN            2)PAYINFO
 3)ADDRESS             4)SALINFO




 =>
    Enter the number of the child you want
    Enter 0 to stay at parent.
```

Enter the number 4. The following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :0

EMP_ID : 112847612

         PAY_DATE      GROSS
         --------      -----

==       820831        1100.00
==       820730        1100.00
==       820630        1100.00
==       820528        1100.00
==       820430        1100.00
==       820331        1100.00
==       820226        1100.00
==       820129        1100.00

-------------------------------INPUT---------------------------------
==
==>
```

Note that the header displays the key field value of the parent instance. Since EMP_ID is the key field of the root segment, the header displays Mary Smith's employee ID.

Also, you could have gone directly from the EMPLOYEE root segment to the monthly pay segment by doing one of the following:

- Typing CHILD 4 on the command line.
- Typing C4 in the prefix area.

Syntax: How to Display the Parent Segment

To return to the parent segment, enter

```
Parent
```

or press PF4 or PF16. The current instance in the parent is the same as before you entered the CHILD command or C prefix area command.

Syntax: How to Display the First Child of the Next Parent Instance

To move to the first child of the next parent instance, enter

```
JUMP
```

or press PF12 or PF24 while FSCAN is displaying a child segment.

Example: Displaying the First Child of the Next Parent Instance

For example, if you enter JUMP while the PAYINFO segment is being displayed for a particular employee, the PAYINFO segment for the next employee in the EMP_INFO segment is displayed. JUMP may be issued anywhere.

Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands

To display a single instance on the screen, enter:

```
SIngle
```

This places you in SINGLE mode. SINGLE mode enables you to view a single segment instance on one screen. Only the current instance appears, but all its fields appear on one screen (unless it has many fields). You may enter all FSCAN commands on the command line at the bottom of the screen, but there is no prefix area. The key field values appear highlighted.

All FSCAN commands (but not prefix area commands) operate in SINGLE mode, except that only one instance is displayed. In particular, note the following:

- If you enter the FORWARD command in SINGLE mode, FSCAN displays the next instance in the segment. If you enter the BACKWARD command, FSCAN displays the previous instance.
- If you enter the CHILD command, only one child instance appears at one time. If you enter the PARENT command, only the parent instance of the current instance appears on the screen.

You can update and delete an instance in SINGLE mode, but you cannot add another instance.

You remain in SINGLE mode until you enter the command:

```
Multiple
```

MULTIPLE returns you to normal mode, which displays multiple instances at one time.

Example: **Using SINGLE Mode**

For example, this is how Diane Jones' instance looks in SINGLE mode. Note that there is no input area, and that the arrow at the bottom of the screen points to the command line where you can enter commands:

```
FSCAN FILE EMPLOYEEFOCUS A1                CHANGES : 0

     EMP_ID : 117593129     LAST_NAME : JONES
 FIRST_NAME : DIANE         HIRE_DATE : 820501
 DEPARTMENT : MIS            CURR_SAL : 18480.00
CURR_JOBCODE : B03             ED_HRS : 50.00



==>
```

Modifying the Database

You may use FSCAN to modify the database by adding, updating, and deleting segment instances.

Adding New Segment Instances: The "I" Prefix

To add a new segment instance to the segment displayed on the screen, type the instance field values in the input area on the bottom of the screen. You can use the Tab key to jump from field to field. Then type I in the prefix area next to the new instance. When you press Enter, FSCAN adds the instance to the database, displaying it in proper sequence based on its key field values.

If the instance you are typing extends beyond the right margin of the screen, use the scrolling commands discussed in *Scrolling the Screen* on page 400. FSCAN adds the segment instance when you press Enter or enter any command except RIGHT, LEFT, RESET, ?, and QQUIT.

Note:

- FSCAN does not accept new instances with key field values that are the same as another instance.
- FSCAN does not accept new instances with field values that do not conform to the ACCEPT attribute in the Master File (ACCEPT is explained in the *Describing Data* manual).

- If you want the new instance to become the current instance, type I/ in the prefix area next to the new instance before pressing Enter.

Example: Adding New Segment Instances

For example, suppose you want to add Fred Johnson to the EMPLOYEE database, and you want the new instance to become the current instance. Type his instance in the input area as shown below (note the I/ in the prefix area):

```
FSCAN FILE EMPLOYEEFOCUS A1     CHANGES :0
     EMP_ID     LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
     ------     ---------  ----------  ---------  ----------
==   117593129  JONES      DIANE       820501     MIS
==   119265415  SMITH      RICHARD     820104     PRODUCTION
==   119329144  BANNING    JOHN        820801     PRODUCTION
==   123764317  IRVING     JOAN        820104     PRODUCTION
==   126724188  ROMANS     ANTHONY     820701     PRODUCTION
==   219984371  MCCOY      JOHN        810701     MIS
==   326179357  BLACKWOOD  ROSEMARIE   820401     MIS
==   451123478  MCKNIGHT   ROGER       820202     PRODUCTION
==   543729165  GREENSPAN  MARY        820401     MIS
==   818692173  CROSS      BARBARA     811102     MIS
----------------------------INPUT-------------------------------------
I/   123123123  johnson    fred        870507     mis

==>
                                                              MORE=>
```

When you press Enter, the screen appears as follows:

```
FSCAN FILE EMPLOYEEFOCUS A1          CHANGES :1

     EMP_ID     LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
     ------     ---------  ---------   ---------  ----------
==   123123123  JOHNSON    FRED        870507      MIS
==   123764317  IRVING     JOAN        820104      PRODUCTION
==   126724188  ROMANS     ANTHONY     820701      PRODUCTION
==   219984371  MCCOY      JOHN        810701      MIS
==   326179357  BLACKWOOD  ROSEMARIE   820401      MIS
==   451123478  MCKNIGHT   ROGER       820202      PRODUCTION
==   543729165  GREENSPAN  MARY        820401      MIS
==   818692173  CROSS      BARBARA     811102      MIS
------------------------------INPUT-----------------------------------
==

==>
  0 Keys Changed 0 Non-Keys Changed
  0 Records Deleted 1 Records Input

                                                                MORE=>
```

If you do not type "I" in the prefix area when you input a new instance, FSCAN displays an error message. To continue, you must do one of the following:

- Enter "I" in the prefix area of the input area.
- Cancel the input by entering the RESET command, typing R in the prefix area, or pressing the PF2 or PF14 key. This also recovers typed-over field values (see the following section).

Note that the RESET command entered on the command line is an immediate command. However, the R prefix-area command is not an immediate command. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Updating Non-Key Field Values

There are three ways to update non-key field values:

- Type over field values.
- Issue the REPLACE command.
- Issue the CHANGE command.

Note that FSCAN does not accept any new field value that does not conform to the ACCEPT attribute in the Master File (the ACCEPT attribute is explained in the *Describing Data* manual).

Procedure: How to Type Over Field Values

You may update segment instances by typing over their values on the screen. Use the Tab key to jump from field to field within the same instance.

Example: Typing Over Field Values

For example, suppose you want to change Richard Smith's department from Production to Sales. Simply type over the DEPARTMENT value and press *Enter*. The screen appears as shown on the next page. Note that the message at the bottom of the screen indicates one changed non-key field.

The screen is:

```
FSCAN FILE EMPLOYEEFOCUS A1              CHANGES :0

      EMP_ID     LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
      ------     ---------  ----------  ---------  ----------
==    071382660  STEVENS    ALFRED      800602     PRODUCTION
==    112847612  SMITH      MARY        810701     MIS
==    117593129  JONES      DIANE       820501     MIS
==    119265415  SMITH      RICHARD     820104     SALES
==    119329144  BANNING    JOHN        820801     PRODUCTION
==    123764317  IRVING     JOAN        820104     PRODUCTION
==    126724188  ROMANS     ANTHONY     820701     PRODUCTION
==    219984371  MCCOY      JOHN        810701     MIS
==    326179357  BLACKWOOD  ROSEMARIE   820401     MIS
==    451123478  MCKNIGHT   ROGER       820202     PRODUCTION
--------------------------------INPUT---------------------------------
==
==>
  0 Keys Changed 1 Non-Keys Changed
  0 Records Deleted 0 Records Input

                                                              MORE=>
```

The message at the bottom of the screen indicates the number of field values you changed since the last time you pressed Enter. The counter at the top of the screen counts the total number of values you changed since the last time the changes were saved on disk.

If you type over field values and change your mind before you press Enter, you can restore the original field values by entering R (to specify the RESET command) on the prefix area next to the instance whose values you are recovering, or by pressing the PF2 or PF14 key. However, if you press Enter before pressing one of these keys, you will not recover the typed-over values.

Note that the RESET command entered on the command line is an immediate command. However, the R prefix area command is not an immediate command. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Syntax: How to Replace Field Values: The REPLACE Command

The REPLACE command replaces one field value with another either for a specific instance or for all the instances in a group. (In the root segment, this is all the instances in the segment; in a descendant segment, this is all the instances that share a parent instance.) The syntax is

```
REPlace field1 = value1[,field2 = value2, ...] [,$ {*|n}]
```

where:

fieldn ...

Is a field in the current instance whose value you want to change.

valuen ...

Is a new value for the field.

,$ {*|*n*}

Enables you to change multiple instances starting from the current instance (the current instance included). *n* is the number of instances to be searched for the field value you want to change. If you want all instances in the group starting from the current instance changed, use an asterisk (*).

Example: Using REPLACE

For example, to change Richard Smith's department from Production to Sales, make Richard Smith's instance the current instance. Then enter:

```
REPLACE DEPARTMENT = SALES
```

To change the DEPARTMENT value to SALES in the next five instances, enter:

```
REPLACE DEPARTMENT = SALES,$ 5
```

To change all DEPARTMENT values in the group to SALES, make the first instance on display the current instance by entering:

```
FIRST
```

Then enter:

```
REPLACE DEPARTMENT = SALES,$ *
```

Syntax: How to Change Character Strings Within Field Values: The CHANGE Command

The CHANGE command changes character strings within field values either for a specific instance or for all the instances in a group (in the root segment, this is all the instances in the segment; in a descendant segment, this is all the instances that share a parent instance). The fields must be alphanumeric. The syntax is

```
CHAnge field = /oldstring/newstring/ [,$ {*|n}]
```

where:

field

Is the name of the field in the current instance whose value you want to change. The field must be alphanumeric, and it cannot be a key field.

oldstring

Is the substring of the field value that you want to change.

newstring

Is the character string to replace the substring.

,$ {*|*n*}

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the substring. If you want all instances in the group searched, starting from the current instance, use an asterisk (*).

Example: Using CHANGE

For example, to change Joan Irving's department from Production to Products, make Joan Irving's instance the current instance. Then enter:

```
CHANGE DEPARTMENT = /ION/S/
```

To change the Production department to Products in the next five instances starting from the current instance, enter:

```
CHANGE DEPARTMENT = /ION/S/,$ 5
```

To change this substring in all the instances in the group, make the first instance on display the current instance by entering:

```
FIRST
```

Then enter:

```
CHANGE DEPARTMENT = /ION/S/ ,$ *
```

Changing Key Field Values

FSCAN enables you to change values of key fields, either by typing over the values or by using the REPLACE KEY command.

Note: FSCAN does not allow you to change a key field to a value that will make the key field values of one instance the same as another instance.

FSCAN does not accept any new key field value that does not conform to the ACCEPT attribute in the Master File (the ACCEPT attribute is explained in the *Describing Data* manual).

Procedure: How to Type Over Key Field Values: The KEY Command

To change the value of a key field, do the following:

1. Type the new value over the old one.
2. Either type a K in the prefix area next to the instance you are changing, or type the command:

   ```
   Key
   ```

   If you want the instance to be the current instance after its key value is changed, type K/ in the prefix area next to the instance.
3. Press Enter.

After you change the key value, FOCUS moves the instance within the segment so that the key values remain sorted in their proper sequence. The screen shows this immediately.

Note: FOCUS does not physically move instances in the root segment, although the instances appear on the FSCAN screen sorted by their key field values.

If you do not enter the KEY command or type K in the prefix area when you change a key field value, FSCAN displays an error message. Before continuing, you must do one of the following:

- Enter the KEY command, or enter K in the prefix area.
- Retype the original key value.
- Restore the key field value by entering the RESET command, typing R in the prefix area, or pressing the PF2 or PF14 key. Other field values you typed over will also be restored.

Note: The RESET command entered on the command line is an immediate command. However, the R prefix area command is not an immediate command. If you type any changes on a line that does not specify the R prefix, FSCAN enters the changes.

Example: Using KEY

For example, suppose you want to change Alfred Stevens' employee ID from 071382660 to 444555666, and you want his instance to remain the current instance. Type over the employee ID and type K/ in the prefix area.

The screen appears as shown below:

```
FSCAN FILE EMPLOYEEFOCUS A1             CHANGES :2

     EMP_ID    LAST_NAME  FIRST_NAME  HIRE_DATE  DEPARTMENT
     ------    ---------  ----------  ---------  ----------
k/   444555666 STEVENS    ALFRED      800602     PRODUCTION
==   112847612 SMITH      MARY        810701     MIS
==   117593129 JONES      DIANE       820501     MIS
==   119265415 SMITH      RICHARD     820104     PRODUCTION
==   119329144 BANNING    JOHN        820801     PRODUCTION
==   123764317 IRVING     JOAN        820104     PRODUCTION
==   126724188 ROMANS     ANTHONY     820701     PRODUCTION
==   219984371 MCCOY      JOHN        810701     MIS
==   326179357 BLACKWOOD  ROSEMARIE   820401     MIS
==   451123478 MCKNIGHT   ROGER       820202     PRODUCTION
-------------------------------INPUT-----------------------------------
==
==>
                                                              MORE=>
```

When you press *Enter*, the screen appears as shown below:

```
FSCAN FILE EMPLOYEEFOCUS A1             CHANGES :3

     EMP_ID    LAST_NAME FIRST_NAME HIRE_DATE DEPARTMENT
     ------    --------- ---------- --------- ----------
==   444555666 STEVENS   ALFRED     800602    PRODUCTION
==   451123478 MCKNIGHT  ROGER      820202    PRODUCTION
==   543729165 GREENSPAN MARY       820401    MIS
==   818692173 CROSS     BARBARA    811102    MIS
-------------------------------INPUT-----------------------------------
==
==>
  1 Keys Changed 0 Non-Keys Changed
  0 Records Deleted 0 Records Input

                                                              MORE=>
```

The message at the bottom of the screen indicates the number of key field values you changed since the last time you pressed *Enter*.

Syntax: How to Change Key Field Values Using the REPLACE KEY Command

You may also use the REPLACE command to change key fields of the current instance. The syntax of the REPLACE command to replace key fields is

```
REPlace KEY key1 = value1[, key2 = value2, ...]
```

where:

keyn ...

Is the key field you want to change. Remember that an instance may have more than one key field (as determined by the SEGTYPE attribute in the Master File).

valuen ...

Is the new value for the key field.

Example: Using REPLACE KEY

For example, to change Alfred Stevens' employee ID from 444555666 to 071382660, make his instance the current instance by placing a slash in the prefix area, and enter the following:

```
REPLACE KEY EMP_ID = 071382660
```

Deleting Segment Instances: The DELETE Command

You can easily delete a data instance with the DELETE command.

Syntax: How to Delete Segment Instances

To delete the current instance, type a D in the prefix area next to the instance or enter:

```
DElete
```

FSCAN displays the complete segment instance alone on the screen and asks if you really want to delete it. Press Enter to delete the instance, or respond:

`N`

Do not delete the current instance. (Returns to the previous screen.)

`Q`

Do not delete the current instance. (If you made no other changes to the database, entering Q leaves FSCAN and returns to the FOCUS prompt. Otherwise, it returns to the previous screen.)

Note: When you delete an instance, you delete all its descendant instances as well.

Example: Using DELETE

For example, suppose you want to delete information about John Banning from the database. First, make John Banning's instance the current instance. Then, enter the DELETE command. The following screen appears:

```
FSCAN FILE EMPLOYEEFOCUS A1            CHANGES :4

            Delete Confirmation Screen

           EMP_ID : 119329144          LAST_NAME : BANNING
       FIRST_NAME : JOHN               HIRE_DATE : 820801
       DEPARTMENT : PRODUCTION          CURR_SAL : 29700.00
     CURR_JOBCODE : A17                    ED_HR :    .00



==>
      Press ENTER to delete
      Enter N(o) to abort
      Enter Q(uit) to quit session
```

If you press Enter, the screen appears as follows:

```
FSCAN FILE EMPLOYEEFOCUS A1CHANGES :2

     EMP_ID    LAST_NAME FIRST_NAME HIRE_DATE DEPARTMENT
     ------    --------- ---------- --------- ----------
==   123764317 IRVING    JOAN       820104    PRODUCTION
==   126724188 ROMANS    ANTHONY    820701    PRODUCTION
==   219984371 MCCOY     JOHN       810701    MIS
==   326179357 BLACKWOOD ROSEMARIE  820401    MIS
==   451123478 MCKNIGHT  ROGER      820202    PRODUCTION
==   543729165 GREENSPAN MARY       820401    MIS
==   818692173 CROSS     BARBARA    811102    MIS
-------------------------------INPUT-----------------------------------
==
==>

0 Keys Changed 0 Non-Keys Changed
1 Records Deleted 0 Records Input

                                                              MORE=>
```

Repeating a Command: ? and =

Two commands help you enter a command repeatedly:

- The ? command displays the last command you entered.
- The = command executes the last command you entered.

Syntax: How to Display Previous Commands: The ? Command

To display the last command you entered, enter

```
?
```

or press *PF6* or *PF18*. This displays the previous command on the command line. You may then execute the command by pressing Enter or remove the command from the command line.

As you enter FSCAN commands on the command line, FSCAN stores them in a stack in memory. If you enter the ? command repeatedly, FSCAN scrolls through the stack, displaying the commands in stack from the most recent to the oldest.

The ? command is an immediate command. The database remains unchanged until you press Enter a second time or enter a non-immediate command. Immediate commands were explained previously at the beginning of *Using FSCAN* on page 394.

Syntax: How to Executing the Previous Command: The = Command

The = command executes the last command you entered. Enter

```
=
```

or press *PF9* or *PF21*.

Saving Changes: The SAVE Without Exiting FSCAN Command

To save the changes to the database that you made on FSCAN, enter

```
SAve
```

You remain in FSCAN. The counter at the top of the screen that counts changes in the database is reset to 0.

Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands

To exit FSCAN and save the changes you made to the database, enter

```
End
```

or:

```
FILe
```

If bad data is encountered upon trying to save your changes, an error message is generated.

To exit FSCAN without saving the changes you made to the database, enter:

```
QQuit
```

Note: QQUIT only suppresses changes made on FSCAN when you are using the Absolute File Integrity facility. Otherwise, FSCAN writes all changes to the database.

If you did not make any changes to the database, you can exit FSCAN by entering

```
QUit
```

or by pressing the *PF3* or *PF15* key.

The FSCAN HELP Facility

FSCAN has a HELP facility. To use HELP, enter the command

```
Help
```

or press the *PF1* or *PF13* key. HELP displays a summary of FSCAN commands and prefix area commands, as shown in the sample screen below:

```
FSCAN FILE CAR     FOCUS A        HELP SCREEN 2 of 3

                        FSCAN COMMANDS
=          - Re-execute the most recent command line.
?          - Retrieve the previous command line.
Backward   - Go backward one screen.
CHAnge     - Change a string within a field:
             CHANGE fieldname=/oldstring/newstring/,$
CHIld      - Display child instances of this segment.
DElete     - Delete a segment instance, and all of its children.
DIsplay    - Display the segment containing the specified fieldname.
End/FILe   - Save all changes and exit FSCAN.
FINd       - Find an instance on this chain which satisfies a test:
             FIND fieldname EQ GT CO... value.
FIRst      - Go to the first instance on this chain.
FOrward    - Go forward one screen.
Jump       - Jump to the children of the next parent.
LAst       - Go to the last instance on this chain
LEft       - Go left one panel.
LOcate     - Same as FIND but search is throughout the database.

Exit HELP: PF03/PF15. Forward: PF08/PF20. Backward: PF07/PF19.
```

You can scroll HELP screens back and forth by pressing the *PF8*or *PF20* key to go forward and the *PF7* or *PF19* key to go backward.

To exit the HELP facility, press the *PF3* or *PF15* key.

Syntax Summary

This section is a summary of the FSCAN commands, PF keys, and prefix area commands. References to other sections are included.

Summary of Commands

FSCAN commands are listed here in alphabetical order. The unique truncation of each command is capitalized.

Backward

Scrolls the display one screen backward.

PF keys: PF7 or PF19.

CHAnge

Changes character strings within field values. The syntax is

```
CHAnge field =/oldstring/newstring/ [,$ {*|n}]
```

where:

field

Is the name of the field whose value you want to change. The field must be alphanumeric and it cannot be a key field.

oldstring

Is the substring of the field value that you want to change.

newstring

Is the character string to replace the substring.

,$ {*|n}

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the substring. If you want all instances in the group searched (starting from the current instance), use an asterisk (*).

You can also change field values by typing over them.

CHIld

Displays the child instances relating to the current instance. (In SINGLE mode, displays the first child instance of the current instance.) The syntax is

`CHIld [n]`

where:

n

Is the number of the child segment as assigned by FSCAN. If you omit this number, FSCAN displays a menu listing the segments and their numbers. Enter a number to display the segment (*Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 411).

Prefix area command: `C[n]`

where:

n

Is the number of the child segment as assigned by FSCAN. If you omit this number, FSCAN displays the menu.

DElete

Deletes the current instance and all descendant instances.

Prefix area command: `D`

DOwn [n]

Scrolls the display *n* lines forward. *n* defaults to 1.

DIsplay Field Name

Displays the segment containing the specified field name.

End

Saves all changes made to the database and exits the FSCAN facility (see *Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands* on page 425).

FILe

Saves all changes made to the database and exits the FSCAN facility (see *Exiting FSCAN: The END, FILE, QQUIT, and QUIT Commands* on page 425).

FINd

Searches a group of instances (in the root segment, this is all instances in the segment; in descendant segments, this is all instances sharing a common parent instance) for an instance containing field values that fulfill certain conditions. FIND searches the group starting from the current instance. If it finds the instance, it makes that instance the current instance.

The syntax is (entered on one line)

```
FINd field1 rel1 value1 [OR value1a OR value1b OR ...]
[{AND|,} field2 rel2 value2 {AND|,}]
```

where:

fieldn ...

Is a field in the segment.

reln ...

Is one of the following relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
CONTAINS or CO	Contains the character string

OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance having the field value that passes the test becomes the current segment. If there are multiple tests, the first instance that passes all the tests becomes the current instance.

OR

Allows you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in the same FIND command.

FIrst

Selects the first instance in a group of instances on display to be the current instance. In the root segment, the group of instances consists of all instances in the segment; in a descendant segment, a group consists of all instances that share a common parent instance.

FOrward

Scrolls the display one screen forward.

PF keys: PF8 or PF20.

Help

Invokes the FSCAN HELP facility.

PF keys: PF01 or PF11.

Input

Adds a new segment instance.

Prefix area command: I

Note: This command is valid only in the input area as a prefix command.

Jump

Moves to the child of the next parent instance.

PF keys: PF12 or PF24

LAst

Selects the last instance of a group of instances on display. In the root segment, the group of instances consists of all instances in the segment; in a descendant segment, a group consists of all instances that share a common parent instance.

LEft

Scrolls the display one panel to the left.

PF keys: PF10 or PF22.

LOcate

Searches for instances containing field values that fulfill certain conditions. LOCATE searches starting from the current instance. If it finds the instance, it makes that instance the current instance.

The syntax is (entered on one line)

```
LOcate field1 rel1 value1 [OR value1a OR value1b OR ...]
[{AND|,} field2 rel2 value2 {AND|,}]
```

where:

fieldn ...

Is a field to be tested.

reln ...

Is one of the following relations:

EQ or =	Equal to
NE	Not equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to

LT	Less than
CONTAINS or CO	Contains the character string
OMITS or OM	Omits the character string

valuen ...

Is a value for which FSCAN can test. The first instance having the field value that passes the test becomes the current segment. If there are multiple tests, the first instance that passes all the tests becomes the current instance.

OR

Allows you to test a field for multiple values. If the field contains one of the values, it meets the test. You can use AND and OR in the same LOCATE command.

Key

Enables you to type over key field values in the current instance.

Prefix area command: K

where:

K/

Makes the instance the current instance after the key values are changed.

Multiple

Displays multiple instances, each on a single line. Entering this command after entering the SINGLE command returns the screen to the normal display (see *Displaying a Single Instance on One Screen: The SINGLE and MULTIPLE Commands* on page 414).

Next [n]

Scrolls the display *n* lines forward. *n* defaults to 1.

Parent

Displays the parent segment. The parent instance becomes the current instance. In SINGLE mode, PARENT displays the parent instance only.

QUit

Exits the FSCAN facility if you did not make any changes to the database.

PF keys: PF3 or PF15.

QQuit

Exits the FSCAN facility without saving any changes to the database.

REPlace

Replaces field values. The syntax is

```
REPlace field1 = value1[,field2 = value2 ...] [,$ {*|n}]
```

where:

fieldn...

Is a field in the instance whose value you want to change.

valuen...

Is the new value for the field.

,$ {*|*n*}

Enables you to change multiple instances counting from the current instance (the current instance included). *n* is the number of instances to be searched for the field values you want to change. If you want all instances in the group searched (starting from the current instance), use an asterisk (*).

You can also replace field values by typing over them.

REPlace KEY

Replaces key field values in the current instance. The syntax is

```
REPlace KEY key1 = value1[, key2 = value2, ...]
```

where:

keyn ...

Is a key field in the instance whose value you want to change.

valuen ...

Is the new value for the key field.

You can also replace key field values by typing over them.

RESet

Performs the following:

- Clears the input area.
- Recovers all field values on the screen that you typed over, both non-key fields and key fields. To recover non-key field values, you must enter the RESET command before you press the *Enter* key. Otherwise, you will not recover the typed-over values.

PF keys: PF2 or PF14.

Prefix area command: R

Note: The R prefix-area command recovers only field values on the line that it is typed. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

RIght

Scrolls the display one panel to the right.

PF keys: PF11 or PF23.

SAve

Saves all changes made to the database without exiting FSCAN.

SIngle

Displays the current instance alone with all field values on one screen. To return to the normal display, enter the MULTIPLE command.

Top

Displays the root segment and makes the first instance in the root segment the current instance, scrolled to the leftmost panel.

?

Displays the previous command in stack.

PF keys: PF6 or PF18.

=

Executes the previous command entered.

PF key: PF9 or PF21.

Summary of PF Keys

The following table is a list of FSCAN PF keys and their corresponding functions.

FSCAN Keys	Functions
PF1, PF13	HELP
PF2, PF14	RESET
PF3, PF15	QUIT
PF4, PF16	PARENT
PF5, PF17	CHILD
PF6, PF18	?
PF7, PF19	BACKWARD
PF8, PF20	FORWARD
PF9, PF21	=
PF10, PF22	LEFT
PF11, PF23	RIGHT
PF12, PF24	JUMP

Summary of Prefix Area Commands

The following is a summary of prefix area commands. You type these commands in the prefix area that corresponds to the instance you wish to address.

/	Makes the instance the current instance. May be typed after the prefix area commands K, I, and R.

C	Displays child instances (see *Displaying Descendant Segments: The CHILD, PARENT, and JUMP Commands* on page 411).
D	Deletes the instance and all its children.
I	Inputs a new instance (valid only in the input area).
I/	Inputs a new instance and makes the instance the current instance (valid only in the input area).
K	Enables you to type over key field values in the instance.
K/	Enables you to type over key field values in the instance, then makes the instance the current instance.
R	Performs the following: ❏ Clears the input area. ❏ Recovers all field values on the screen that you typed over, both non-key fields and key fields. To recover non-key field values, you must enter the RESET command before you press the Enter key. Otherwise, you will not recover the typed-over values. Note that the R prefix area command recovers only field values on the line on which it is typed. If you typed changes on a line not specifying the R prefix, FSCAN enters the changes.

Chapter 6

Master Files and Diagrams

This appendix contains descriptions and structure diagrams for the sample data sources used throughout the documentation.

In this chapter:

Creating Sample Data Sources

Create sample data sources on your user ID by executing the procedures specified below. These FOCEXECs are supplied with FOCUS. If they are not available to you or if they produce error messages, contact your systems administrator or Information Builders Customer Support Services.

To create these files, first make sure you have read access to the Master Files.

Data Source	Load Procedure Name
EMPLOYEE, EDUCFILE, and JOBFILE	`EX EMPTSO` These FOCEXECs also test the data sources by generating sample reports. If you are using Hot Screen, remember to press either Enter or the PF3 key after each report. If the EMPLOYEE, EDUCFILE, and JOBFILE data sources already exist on your user ID, the FOCEXEC replaces them with new copies. This FOCEXEC assumes that the high-level qualifier for the FOCUS data sources is the same as the high-level qualifier for the MASTER PDS that was unloaded from the tape.
SALES PROD	`EX SALES` `EX PROD`
CAR	None (created automatically during installation).

LEDGER
FINANCE
REGION
COURSES
EXPERSON
	`EX LEDGER`
	`EX FINANCE`
	`EX REGION`
	`EX COURSES`
	`EX EXPERSON`

EMPDATA TRAINING COURSE JOBHIST JOBLIST LOCATOR PERSINFO SALHIST	`EX LOADPERS`

Data Source	Load Procedure Name
PAYHIST	None (PAYHIST DATA is a sequential data source and is allocated during the installation process).
COMASTER	None (COMASTER is used for debugging other Master Files).
VIDEOTRK and MOVIES	`EX LOADVTRK`
VIDEOTR2	`EX LOADVID2`
Gotham Grinds	`EX DBLGG`

Century Corp:
CENTCOMP
CENTFIN
CENTHR
CENTINV
CENTORD
CENTQA
CENTGL
CENTSYSF
CENTSTMT
	
	`EX LOADCOM`
	`EX LOADFIN`
	`EX LOADHR`
	`EX LOADINV`
	`EX LOADORD`
	`EX LOADCQA`
	`EX LDCENTGL`
	`EX LDCENTSY`
	`EX LDSTMT`

EMPLOYEE Data Source

EMPLOYEE contains sample data about company employees. Its segments are:

`EMPINFO`

Contains employee IDs, names, and positions.

`FUNDTRAN`

Specifies employee direct deposit accounts. This segment is unique.

`PAYINFO`

Contains the employee salary history.

`ADDRESS`

Contains employee home and bank addresses.

`SALINFO`

Contains data on employee monthly pay.

`DEDUCT`

Contains data on monthly pay deductions.

EMPLOYEE also contains cross-referenced segments belonging to the JOBFILE and EDUCFILE files, also described in this appendix. The segments are:

`JOBSEG (from JOBFILE)`

Describes the job positions held by each employee.

`SKILLSEG (from JOBFILE)`

Lists the skills required by each position.

`SECSEG (from JOBFILE)`

Specifies the security clearance needed for each job position.

`ATTNDSEG (from EDUCFILE)`

Lists the dates that employees attended in-house courses.

```
COURSEG (from EDUCFILE)
```

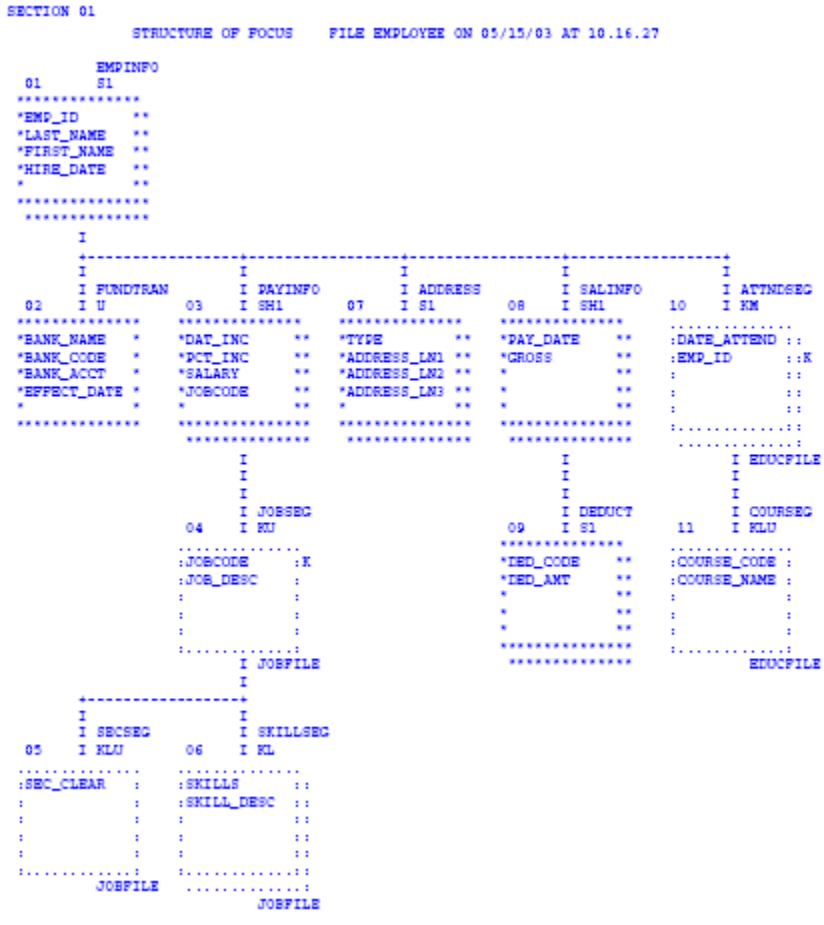
Lists the courses that the employees attended.

EMPLOYEE Master File

```
FILENAME=EMPLOYEE, SUFFIX=FOC
 SEGNAME=EMPINFO,  SEGTYPE=S1
  FIELDNAME=EMP_ID,         ALIAS=EID,     FORMAT=A9,           $
  FIELDNAME=LAST_NAME,      ALIAS=LN,      FORMAT=A15,          $
  FIELDNAME=FIRST_NAME,     ALIAS=FN,      FORMAT=A10,          $
  FIELDNAME=HIRE_DATE,      ALIAS=HDT,     FORMAT=I6YMD,        $
  FIELDNAME=DEPARTMENT,     ALIAS=DPT,     FORMAT=A10,          $
  FIELDNAME=CURR_SAL,       ALIAS=CSAL,    FORMAT=D12.2M,       $
  FIELDNAME=CURR_JOBCODE,   ALIAS=CJC,     FORMAT=A3,           $
  FIELDNAME=ED_HRS,         ALIAS=OJT,     FORMAT=F6.2,         $
 SEGNAME=FUNDTRAN,  SEGTYPE=U,   PARENT=EMPINFO
  FIELDNAME=BANK_NAME,      ALIAS=BN,      FORMAT=A20,          $
  FIELDNAME=BANK_CODE,      ALIAS=BC,      FORMAT=I6S,          $
  FIELDNAME=BANK_ACCT,      ALIAS=BA,      FORMAT=I9S,          $
  FIELDNAME=EFFECT_DATE,    ALIAS=EDATE,   FORMAT=I6YMD,        $
 SEGNAME=PAYINFO,   SEGTYPE=SH1, PARENT=EMPINFO
  FIELDNAME=DAT_INC,        ALIAS=DI,      FORMAT=I6YMD,        $
  FIELDNAME=PCT_INC,        ALIAS=PI,      FORMAT=F6.2,         $
  FIELDNAME=SALARY,         ALIAS=SAL,     FORMAT=D12.2M,       $
  FIELDNAME=JOBCODE,        ALIAS=JBC,     FORMAT=A3,           $
 SEGNAME=ADDRESS,   SEGTYPE=S1,  PARENT=EMPINFO
  FIELDNAME=TYPE,           ALIAS=AT,      FORMAT=A4,           $
  FIELDNAME=ADDRESS_LN1,    ALIAS=LN1,     FORMAT=A20,          $
  FIELDNAME=ADDRESS_LN2,    ALIAS=LN2,     FORMAT=A20,          $
  FIELDNAME=ADDRESS_LN3,    ALIAS=LN3,     FORMAT=A20,          $
  FIELDNAME=ACCTNUMBER,     ALIAS=ANO,     FORMAT=I9L,          $
 SEGNAME=SALINFO,   SEGTYPE=SH1, PARENT=EMPINFO
  FIELDNAME=PAY_DATE,       ALIAS=PD,      FORMAT=I6YMD,        $
  FIELDNAME=GROSS,          ALIAS=MO_PAY,  FORMAT=D12.2M,       $
 SEGNAME=DEDUCT,    SEGTYPE=S1,  PARENT=SALINFO
  FIELDNAME=DED_CODE,       ALIAS=DC,      FORMAT=A4,           $
  FIELDNAME=DED_AMT,        ALIAS=DA,      FORMAT=D12.2M,       $
 SEGNAME=JOBSEG,    SEGTYPE=KU,  PARENT=PAYINFO, CRFILE=JOBFILE,
  CRKEY=JOBCODE,$
 SEGNAME=SECSEG,    SEGTYPE=KLU, PARENT=JOBSEG,   CRFILE=JOBFILE, $
 SEGNAME=SKILLSEG,  SEGTYPE=KL,  PARENT=JOBSEG,   CRFILE=JOBFILE, $
 SEGNAME=ATTNDSEG,  SEGTYPE=KM,  PARENT=EMPINFO,  CRFILE=EDUCFILE,
  CRKEY=EMP_ID,$
 SEGNAME=COURSEG,   SEGTYPE=KLU, PARENT=ATTNDSEG, CRFILE=EDUCFILE,$
```

EMPLOYEE Structure Diagram

The EMPLOYEE structure follows:



JOBFILE Data Source

JOBFILE contains sample data about company job positions. Its segments are:

JOBSEG

Describes what each position is. The field JOBCODE in this segment is indexed.

SKILLSEG

Lists the skills required by each position.

SECSEG

Specifies the security clearance needed, if any. This segment is unique.

JOBFILE Master File

```
FILENAME=JOBFILE,  SUFFIX=FOC
 SEGNAME=JOBSEG,   SEGTYPE=S1
  FIELDNAME=JOBCODE,    ALIAS=JC, FORMAT=A3,    INDEX=I,$
  FIELDNAME=JOB_DESC,   ALIAS=JD, FORMAT=A25           ,$
 SEGNAME=SKILLSEG, SEGTYPE=S1,  PARENT=JOBSEG
  FIELDNAME=SKILLS,     ALIAS=,   FORMAT=A4            ,$
  FIELDNAME=SKILL_DESC, ALIAS=SD, FORMAT=A30           ,$
 SEGNAME=SECSEG,   SEGTYPE=U,   PARENT=JOBSEG
  FIELDNAME=SEC_CLEAR,  ALIAS=SC, FORMAT=A6            ,$
```

JOBFILE Structure Diagram

```
SECTION 01
             STRUCTURE OF FOCUS    FILE JOBFILE ON 05/15/03 AT 14.40.06

          JOBSEG
  01      S1
 **************
 *JOBCODE     **I
 *JOB_DESC    **
 *            **
 *            **
 *            **
 ***************
  **************
        I
        +-----------------+
        I                 I
        I SECSEG          I SKILLSEG
  02    I U         03    I S1
 **************    *************
 *SEC_CLEAR   *    *SKILLS     **
 *            *    *SKILL_DESC **
 *            *    *           **
 *            *    *           **
 *            *    *           **
 **************    **************
                    *************
```

EDUCFILE Data Source

EDUCFILE contains sample data about company in-house courses. Its segments are:

`COURSEG`

Contains data on each course.

`ATTNDSEG`

Specifies which employees attended the courses. Both fields in the segment are key fields. The field EMP_ID in this segment is indexed.

EDUCFILE Master File

```
FILENAME=EDUCFILE, SUFFIX=FOC
 SEGNAME=COURSEG,  SEGTYPE=S1
  FIELDNAME=COURSE_CODE,  ALIAS=CC,   FORMAT=A6,          $
  FIELDNAME=COURSE_NAME,  ALIAS=CD,   FORMAT=A30,         $
 SEGNAME=ATTNDSEG,  SEGTYPE=SH2,  PARENT=COURSEG
  FIELDNAME=DATE_ATTEND,  ALIAS=DA,   FORMAT=I6YMD,       $
  FIELDNAME=EMP_ID,       ALIAS=EID,  FORMAT=A9, INDEX=I, $
```

EDUCFILE Structure Diagram

```
SECTION 01
              STRUCTURE OF FOCUS    FILE EDUCFILE ON 05/15/03 AT 14.45.44

          COURSEG
  01      S1
 **************
 *COURSE_CODE **
 *COURSE_NAME **
 *            **
 *            **
 *            **
 ***************
  **************
        I
        I
        I
        I ATTNDSEG
  02    I SH2
 **************
 *DATE_ATTEND **
 *EMP_ID      **I
 *            **
 *            **
 *            **
 ***************
  **************
```

SALES Data Source

SALES contains sample data about a dairy company with an affiliated store chain. Its segments are:

STOR_SEG

Lists the stores buying the products.

DAT_SEG

Contains the dates of inventory.

PRODUCT

Contains sales data for each product on each date. The PROD_CODE field is indexed. The RETURNS and DAMAGED fields have the MISSING=ON attribute.

SALES Master File

```
FILENAME=KSALES,    SUFFIX=FOC
 SEGNAME=STOR_SEG, SEGTYPE=S1
  FIELDNAME=STORE_CODE,  ALIAS=SNO,    FORMAT=A3,    $
  FIELDNAME=CITY,        ALIAS=CTY,    FORMAT=A15,   $
  FIELDNAME=AREA,        ALIAS=LOC,    FORMAT=A1,    $
 SEGNAME=DATE_SEG, PARENT=STOR_SEG, SEGTYPE=SH1,
  FIELDNAME=DATE,        ALIAS=DTE,    FORMAT=A4MD, $
 SEGNAME=PRODUCT,  PARENT=DATE_SEG, SEGTYPE=S1,
  FIELDNAME=PROD_CODE,   ALIAS=PCODE, FORMAT=A3,    FIELDTYPE=I,$
  FIELDNAME=UNIT_SOLD,   ALIAS=SOLD,  FORMAT=I5,    $
  FIELDNAME=RETAIL_PRICE,ALIAS=RP,    FORMAT=D5.2M,$
  FIELDNAME=DELIVER_AMT, ALIAS=SHIP,  FORMAT=I5,    $
  FIELDNAME=OPENING_AMT, ALIAS=INV,   FORMAT=I5,    $
  FIELDNAME=RETURNS,     ALIAS=RTN,   FORMAT=I3,    MISSING=ON,$
  FIELDNAME=DAMAGED,     ALIAS=BAD,   FORMAT=I3,    MISSING=ON,$
```

SALES Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE SALES ON 05/15/03 AT 14.50.28

         STOR_SEG
 01      S1
 **************
 *STORE_CODE  **
 *CITY        **
 *AREA        **
 *            **
 *            **
 ***************
  ***************
       I
       I
       I
       I DATE_SEG
 02    I SH1
 **************
 *DATE        **
 *            **
 *            **
 *            **
 *            **
 ***************
  ***************
       I
       I
       I
       I PRODUCT
 03    I S1
 **************
 *PROD_CODE   **I
 *UNIT_SOLD   **
 *RETAIL_PRICE**
 *DELIVER_AMT **
 *            **
 ***************
  ***************
```

PROD Data Source

The PROD data source lists products sold by a dairy company. It consists of one segment, PRODUCT. The field PROD_CODE is indexed.

PROD Master File

```
FILE=KPROD, SUFFIX=FOC
 SEGMENT=PRODUCT, SEGTYPE=S1,
  FIELDNAME=PROD_CODE, ALIAS=PCODE, FORMAT=A3,    FIELDTYPE=I, $
  FIELDNAME=PROD_NAME, ALIAS=ITEM,  FORMAT=A15,   $
  FIELDNAME=PACKAGE,   ALIAS=SIZE,  FORMAT=A12,   $
  FIELDNAME=UNIT_COST, ALIAS=COST,  FORMAT=D5.2M, $
```

PROD Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS   FILE PROD   ON 05/15/03 AT 14.57.38
       PRODUCT
01     S1
*************
*PROD_CODE   **I
*PROD_NAME   **
*PACKAGE     **
*UNIT_COST   **
*            **
***************
 *************
```

CAR Data Source

CAR contains sample data about specifications and sales information for rare cars. Its segments are:

`ORIGIN`

Lists the country that manufactures the car. The field COUNTRY is indexed.

`COMP`

Contains the car name.

`CARREC`

Contains the car model.

`BODY`

Lists the body type, seats, dealer and retail costs, and units sold.

SPECS

Lists car specifications. This segment is unique.

WARANT

Lists the type of warranty.

EQUIP

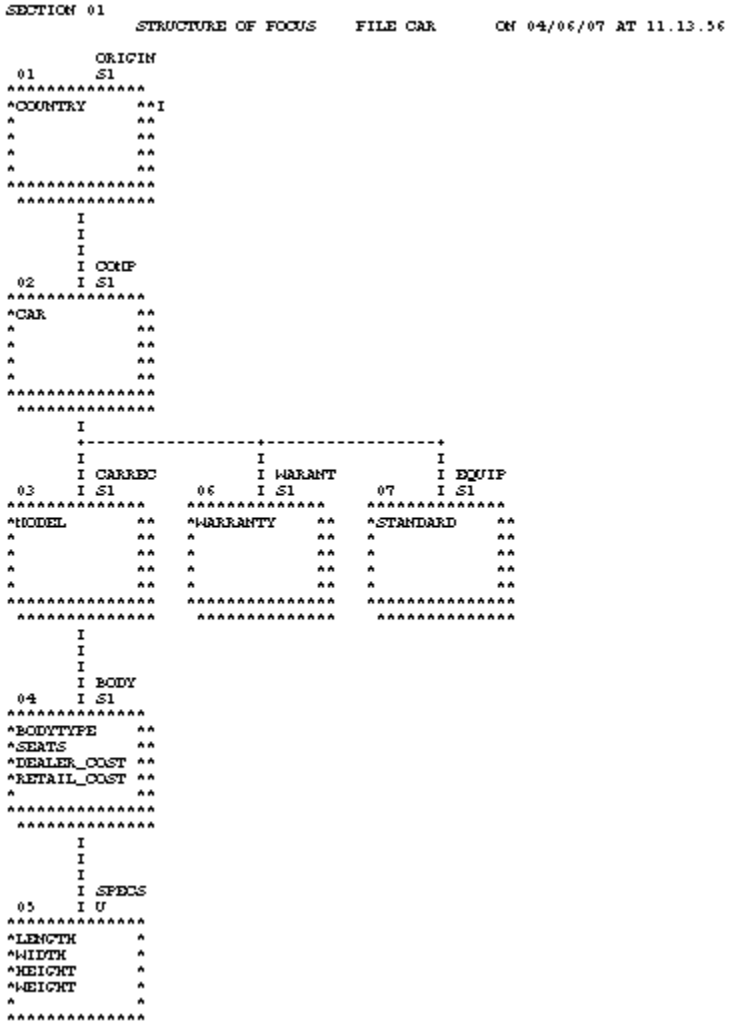
Lists standard equipment.

The aliases in the CAR Master File are specified without the ALIAS keyword.

CAR Master File

```
FILENAME=CAR,SUFFIX=FOC
 SEGNAME=ORIGIN,SEGTYPE=S1
  FIELDNAME=COUNTRY,COUNTRY,A10,FIELDTYPE=I,$
 SEGNAME=COMP,SEGTYPE=S1,PARENT=ORIGIN
  FIELDNAME=CAR,CARS,A16,$
 SEGNAME=CARREC,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=MODEL,MODEL,A24,$
 SEGNAME=BODY,SEGTYPE=S1,PARENT=CARREC
  FIELDNAME=BODYTYPE,TYPE,A12,$
  FIELDNAME=SEATS,SEAT,I3,$
  FIELDNAME=DEALER_COST,DCOST,D7,$
  FIELDNAME=RETAIL_COST,RCOST,D7,$
  FIELDNAME=SALES,UNITS,I6,$
 SEGNAME=SPECS,SEGTYPE=U,PARENT=BODY
  FIELDNAME=LENGTH,LEN,D5,$
  FIELDNAME=WIDTH,WIDTH,D5,$
  FIELDNAME=HEIGHT,HEIGHT,D5,$
  FIELDNAME=WEIGHT,WEIGHT,D6,$
  FIELDNAME=WHEELBASE,BASE,D6.1,$
  FIELDNAME=FUEL_CAP,FUEL,D6.1,$
  FIELDNAME=BHP,POWER,D6,$
  FIELDNAME=RPM,RPM,I5,$
  FIELDNAME=MPG,MILES,D6,$
  FIELDNAME=ACCEL,SECONDS,D6,$
 SEGNAME=WARANT,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=WARRANTY,WARR,A40,$
 SEGNAME=EQUIP,SEGTYPE=S1,PARENT=COMP
  FIELDNAME=STANDARD,EQUIP,A40,$
```

CAR Structure Diagram



LEDGER Data Source

LEDGER contains sample accounting data. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

LEDGER Master File

```
FILENAME=LEDGER, SUFFIX=FOC,$
 SEGNAME=TOP,    SEGTYPE=S2,$
  FIELDNAME=YEAR    ,   , FORMAT=A4, $
  FIELDNAME=ACCOUNT,   , FORMAT=A4, $
  FIELDNAME=AMOUNT ,   , FORMAT=I5C,$
```

LEDGER Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE LEDGER   ON 05/15/03 AT 15.17.08

          TOP
  01      S2
 **************
 *YEAR         **
 *ACCOUNT      **
 *AMOUNT       **
 *             **
 *             **
 ***************
  **************
```

FINANCE Data Source

FINANCE contains sample financial data for balance sheets. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

FINANCE Master File

```
FILENAME=FINANCE, SUFFIX=FOC,$
 SEGNAME=TOP,     SEGTYPE=S2,$
  FIELDNAME=YEAR    ,   , FORMAT=A4,   $
  FIELDNAME=ACCOUNT,   , FORMAT=A4,   $
  FIELDNAME=AMOUNT ,   , FORMAT=D12C,$
```

FINANCE Structure Diagram

```
SECTION 01
           STRUCTURE OF FOCUS    FILE FINANCE  ON 05/15/03 AT 15.17.08

          TOP
 01       S2
 **************
 *YEAR          **
 *ACCOUNT       **
 *AMOUNT        **
 *              **
 *              **
 ***************
  **************
```

REGION Data Source

REGION contains sample account data for the eastern and western regions of the country. It consists of one segment, TOP. This data source is specified primarily for FML examples. Aliases do not exist for the fields in this Master File, and the commas act as placeholders.

REGION Master File

```
FILENAME=REGION, SUFFIX=FOC,$
 SEGNAME=TOP,    SEGTYPE=S1,$
  FIELDNAME=ACCOUNT,  , FORMAT=A4, $
  FIELDNAME=E_ACTUAL, , FORMAT=I5C,$
  FIELDNAME=E_BUDGET, , FORMAT=I5C,$
  FIELDNAME=W_ACTUAL, , FORMAT=I5C,$
  FIELDNAME=W_BUDGET, , FORMAT=I5C,$
```

REGION Structure Diagram

```
SECTION 01
           STRUCTURE OF FOCUS    FILE REGION   ON 05/15/03 AT 15.18.48

          TOP
 01       S1
 ***************
 *ACCOUNT       **
 *E_ACTUAL      **
 *E_BUDGET      **
 *W_ACTUAL      **
 *              **
 ***************
  **************
```

COURSES Data Source

COURSES contains sample data about education courses. It consists of one segment, CRSESEG1. The field DESCRIPTION has a format of TEXT (TX).

COURSES Master File

```
FILENAME=COURSES,  SUFFIX=FOC,$
 SEGNAME=CRSESEG1, SEGTYPE=S1, $
  FIELDNAME=COURSE_CODE,  ALIAS=CC,    FORMAT=A6,   FIELDTYPE=I, $
  FIELDNAME=COURSE_NAME,  ALIAS=CN,    FORMAT=A30,               $
  FIELDNAME=DURATION,     ALIAS=DAYS,  FORMAT=I3,                $
  FIELDNAME=DESCRIPTION,  ALIAS=CDESC, FORMAT=TX50,              $
```

COURSES Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE COURSES   ON 05/15/03 AT 12.26.05

        CRSESEG1
 01     S1
**************
*COURSE_CODE **I
*COURSE_NAME **
*DURATION    **
*DESCRIPTION **T
*            **
***************
 ***************
```

EMPDATA Data Source

EMPDATA contains sample data about company employees. It consists of one segment, EMPDATA. The PIN field is indexed. The AREA field is a temporary field.

EMPDATA Master File

```
FILENAME=EMPDATA, SUFFIX=FOC
 SEGNAME=EMPDATA, SEGTYPE=S1
  FIELDNAME=PIN,          ALIAS=ID,       FORMAT=A9,  INDEX=I,    $
  FIELDNAME=LASTNAME,     ALIAS=LN,       FORMAT=A15,             $
  FIELDNAME=FIRSTNAME,    ALIAS=FN,       FORMAT=A10,             $
  FIELDNAME=MIDINITIAL,   ALIAS=MI,       FORMAT=A1,              $
  FIELDNAME=DIV,          ALIAS=CDIV,     FORMAT=A4,              $
  FIELDNAME=DEPT,         ALIAS=CDEPT,    FORMAT=A20,             $
  FIELDNAME=JOBCLASS,     ALIAS=CJCLAS,   FORMAT=A8,              $
  FIELDNAME=TITLE,        ALIAS=CFUNC,    FORMAT=A20,             $
  FIELDNAME=SALARY,       ALIAS=CSAL,     FORMAT=D12.2M,          $
  FIELDNAME=HIREDATE,     ALIAS=HDAT,     FORMAT=YMD,             $
$
DEFINE AREA/A13=DECODE DIV (NE 'NORTH EASTERN' SE 'SOUTH EASTERN'
CE 'CENTRAL' WE 'WESTERN' CORP 'CORPORATE' ELSE 'INVALID AREA');$
```

EMPDATA Structure Diagram

```
SECTION 01
           STRUCTURE OF FOCUS    FILE EMPDATA  ON 05/15/03 AT 14.49.09

         EMPDATA
  01     S1
 *************
 *PIN         **I
 *LASTNAME    **
 *FIRSTNAME   **
 *MIDINITIAL  **
 *            **
 ***************
  *************
```

EXPERSON Data Source

The EXPERSON data source contains personal data about individual employees. It consists of one segment, ONESEG.

EXPERSON Master File

```
FILE=EXPERSON    ,SUFFIX=FOC
 SEGMENT=ONESEG, $
  FIELDNAME=SOC_SEC_NO   ,ALIAS=SSN        ,USAGE=A9        ,$
  FIELDNAME=FIRST_NAME   ,ALIAS=FN         ,USAGE=A9        ,$
  FIELDNAME=LAST_NAME    ,ALIAS=LN         ,USAGE=A10       ,$
  FIELDNAME=AGE          ,ALIAS=YEARS      ,USAGE=I2        ,$
  FIELDNAME=SEX          ,ALIAS=           ,USAGE=A1        ,$
  FIELDNAME=MARITAL_STAT ,ALIAS=MS         ,USAGE=A1        ,$
  FIELDNAME=NO_DEP       ,ALIAS=NDP        ,USAGE=I3        ,$
  FIELDNAME=DEGREE       ,ALIAS=           ,USAGE=A3        ,$
  FIELDNAME=NO_CARS      ,ALIAS=CARS       ,USAGE=I3        ,$
  FIELDNAME=ADDRESS      ,ALIAS=           ,USAGE=A14       ,$
  FIELDNAME=CITY         ,ALIAS=           ,USAGE=A10       ,$
  FIELDNAME=WAGE         ,ALIAS=PAY        ,USAGE=D10.2SM   ,$
  FIELDNAME=CATEGORY     ,ALIAS=STATUS     ,USAGE=A1        ,$
  FIELDNAME=SKILL_CODE   ,ALIAS=SKILLS     ,USAGE=A5        ,$
  FIELDNAME=DEPT_CODE    ,ALIAS=WHERE      ,USAGE=A4        ,$
  FIELDNAME=TEL_EXT      ,ALIAS=EXT        ,USAGE=I4        ,$
  FIELDNAME=DATE_EMP     ,ALIAS=BASE_DATE  ,USAGE=I6YMTD    ,$
  FIELDNAME=MULTIPLIER   ,ALIAS=RATIO      ,USAGE=D5.3      ,$
```

EXPERSON Structure Diagram

```
SECTION 01
           STRUCTURE OF FOCUS    FILE EXPERSON  ON 05/15/03 AT 14.50.58

         ONESEG
  01     S1
 **************
 *SOC_SEC_NO  **
 *FIRST_NAME  **
 *LAST_NAME   **
 *AGE         **
 *            **
 ***************
  **************
```

TRAINING Data Source

TRAINING contains sample data about training courses for employees. It consists of one segment, TRAINING. The PIN field is indexed. The EXPENSES, GRADE, and LOCATION fields have the MISSING=ON attribute.

TRAINING Master File

```
FILENAME=TRAINING, SUFFIX=FOC
 SEGNAME=TRAINING, SEGTYPE=SH3
  FIELDNAME=PIN,          ALIAS=ID,     FORMAT=A9,   INDEX=I,    $
  FIELDNAME=COURSESTART,  ALIAS=CSTART, FORMAT=YMD,              $
  FIELDNAME=COURSECODE,   ALIAS=CCOD,   FORMAT=A7,               $
  FIELDNAME=EXPENSES,     ALIAS=COST,   FORMAT=D8.2, MISSING=ON  $
  FIELDNAME=GRADE,        ALIAS=GRA,    FORMAT=A2,   MISSING=ON, $
  FIELDNAME=LOCATION,     ALIAS=LOC,    FORMAT=A6,   MISSING=ON, $
```

TRAINING Structure Diagram

```
SECTION 01
           STRUCTURE OF FOCUS    FILE TRAINING ON 05/15/03 AT 14.51.28

         TRAINING
  01     SH3
 **************
 *PIN         **I
 *COURSESTART **
 *COURSECODE  **
 *EXPENSES    **
 *            **
 ***************
  **************
```

COURSE Data Source

COURSE contains sample data about education courses. It consists of one segment, CRSELIST.

COURSE Master File

```
FILENAME=COURSE,   SUFFIX=FOC
 SEGNAME=CRSELIST,  SEGTYPE=S1
  FIELDNAME=COURSECODE,  ALIAS=CCOD,   FORMAT=A7,   INDEX=I,    $
  FIELDNAME=CTITLE,      ALIAS=COURSE, FORMAT=A35,              $
  FIELDNAME=SOURCE,      ALIAS=ORG,    FORMAT=A35,              $
  FIELDNAME=CLASSIF,     ALIAS=CLASS,  FORMAT=A10,              $
  FIELDNAME=TUITION,     ALIAS=FEE,    FORMAT=D8.2, MISSING=ON, $
  FIELDNAME=DURATION,    ALIAS=DAYS,   FORMAT=A3,   MISSING=ON, $
  FIELDNAME=DESCRIPTN1,  ALIAS=DESC1,  FORMAT=A40,              $
  FIELDNAME=DESCRIPTN2,  ALIAS=DESC2,  FORMAT=A40,              $
  FIELDNAME=DESCRIPTN2,  ALIAS=DESC3,  FORMAT=A40,              $
```

COURSE Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE COURSE   ON 05/15/03 AT 12.26.05

         CRSELIST
 01      S1
**************
*COURSECODE  **I
*CTITLE      **
*SOURCE      **
*CLASSIF     **
*            **
***************
 **************
```

JOBHIST Data Source

JOBHIST contains information about employee jobs. Both the PIN and JOBSTART fields are keys. The PIN field is indexed.

JOBHIST Master File

```
FILENAME=JOBHIST, SUFFIX=FOC
SEGNAME=JOBHIST, SEGTYPE=SH2
 FIELDNAME=PIN,          ALIAS=ID,       FORMAT=A9,      INDEX=I ,$
 FIELDNAME=JOBSTART,     ALIAS=SDAT,     FORMAT=YMD,              $
 FIELDNAME=JOBCLASS,     ALIAS=JCLASS,   FORMAT=A8,               $
 FIELDNAME=FUNCTITLE,    ALIAS=FUNC,     FORMAT=A20,              $
```

JOBHIST Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE JOBHIST  ON 01/22/08 AT 16.23.46
    JOBHIST
  01      SH2
 **************
 *PIN         **I
 *JOBSTART    **
 *JOBCLASS    **
 *FUNCTITLE   **
 *            **
 ***************
  **************
```

JOBLIST Data Source

JOBLIST contains information about jobs. The JOBCLASS field is indexed.

JOBLIST Master File

```
FILENAME=JOBLIST, SUFFIX=FOC
SEGNAME=JOBSEG,   SEGTYPE=S1
 FIELDNAME=JOBCLASS,     ALIAS=JCLASS,   FORMAT=A8,    INDEX=I ,$
 FIELDNAME=CATEGORY,     ALIAS=JGROUP,   FORMAT=A25,            $
 FIELDNAME=JOBDESC,      ALIAS=JDESC,    FORMAT=A40,            $
 FIELDNAME=LOWSAL,       ALIAS=LSAL,     FORMAT=D12.2M,         $
 FIELDNAME=HIGHSAL,      ALIAS=HSAL,     FORMAT=D12.2M,         $
DEFINE GRADE/A2=EDIT  (JCLASS,'$$$99');$
DEFINE LEVEL/A25=DECODE GRADE (08 'GRADE 8' 09 'GRADE 9' 10
'GRADE 10' 11 'GRADE 11' 12 'GRADE 12' 13 'GRADE 13' 14 'GRADE 14');$
```

JOBLIST Structure Diagram

```
SECTION 01
             STRUCTURE OF FOCUS    FILE JOBLIST  ON 01/22/08 AT 16.24.52
         JOBSEG
  01     S1
 **************
 *JOBCLASS    **I
 *CATEGORY    **
 *JOBDESC     **
 *LOWSAL      **
 *            **
 ***************
  **************
```

LOCATOR Data Source

JOBHIST contains information about employee location and phone number. The PIN field is indexed.

LOCATOR Master File

```
FILENAME=LOCATOR, SUFFIX=FOC
SEGNAME=LOCATOR,   SEGTYPE=S1,
 FIELDNAME=PIN,           ALIAS=ID_NO,    FORMAT=A9,   INDEX=I, $
 FIELDNAME=SITE,          ALIAS=SITE,     FORMAT=A25,           $
 FIELDNAME=FLOOR,         ALIAS=FL,       FORMAT=A3,            $
 FIELDNAME=ZONE,          ALIAS=ZONE,     FORMAT=A2,            $
 FIELDNAME=BUS_PHONE,     ALIAS=BTEL,     FORMAT=A5,            $
```

LOCATOR Structure Diagram

```
SECTION 01
             STRUCTURE OF FOCUS    FILE LOCATOR  ON 01/22/08 AT 16.26.55
         LOCATOR
  01     S1
 **************
 *PIN          **I
 *SITE         **
 *FLOOR        **
 *ZONE         **
 *             **
 ***************
  **************
```

PERSINFO Data Source

PERSINFO contains employee personal information. The PIN field is indexed.

PERSINFO Master File

```
FILENAME=PERSINFO, SUFFIX=FOC
SEGNAME=PERSONAL, SEGTYPE=S1
 FIELDNAME=PIN,           ALIAS=ID,       FORMAT=A9,   INDEX=I,    $
 FIELDNAME=INCAREOF,      ALIAS=ICO,      FORMAT=A35,              $
 FIELDNAME=STREETNO,      ALIAS=STR,      FORMAT=A20,              $
 FIELDNAME=APT,           ALIAS=APT,      FORMAT=A4,               $
 FIELDNAME=CITY,          ALIAS=CITY,     FORMAT=A20,              $
 FIELDNAME=STATE,         ALIAS=PROV,     FORMAT=A4,               $
 FIELDNAME=POSTALCODE,    ALIAS=ZIP,      FORMAT=A10,              $
 FIELDNAME=COUNTRY,       ALIAS=CTRY,     FORMAT=A15,              $
 FIELDNAME=HOMEPHONE,     ALIAS=TEL,      FORMAT=A10,              $
 FIELDNAME=EMERGENCYNO,   ALIAS=ENO,      FORMAT=A10,              $
 FIELDNAME=EMERGCONTACT,  ALIAS=ENAME,    FORMAT=A35,              $
 FIELDNAME=RELATIONSHIP,  ALIAS=REL,      FORMAT=A8,               $
 FIELDNAME=BIRTHDATE,     ALIAS=BDAT,     FORMAT=YMD,              $
```

PERSINFO Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE PERSINFO ON 01/22/08 AT 16.27.24
   PERSONAL
  01      S1
 **************
 *PIN          **I
 *INCAREOF     **
 *STREETNO     **
 *APT          **
 *             **
 ***************
  **************
```

SALHIST Data Source

SALHIST contains information about employee salary history. The PIN field is indexed. Both the PIN and EFFECTDATE fields are keys.

SALHIST Master File

```
FILENAME=SALHIST,  SUFFIX=FOC
SEGNAME=SLHISTRY,  SEGTYPE=SH2
 FIELDNAME=PIN,        ALIAS=ID,     FORMAT=A9,     INDEX=I,  $
 FIELDNAME=EFFECTDATE, ALIAS=EDAT,   FORMAT=YMD,              $
 FIELDNAME=OLDSALARY,  ALIAS=OSAL,   FORMAT=D12.2,            $
```

SALHIST Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE SALHIST  ON 01/22/08 AT 16.28.02
    SLHISTRY
  01      SH2
 **************
 *PIN         **I
 *EFFECTDATE  **
 *OLDSALARY   **
 *            **
 *            **
 ***************
  **************
```

PAYHIST File

The PAYHIST data source contains the employees' salary history. It consists of one segment, PAYSEG. The SUFFIX attribute indicates that the data file is a fixed-format sequential file.

PAYHIST Master File

```
FILENAME=PAYHIST,  SUFFIX=FIX
 SEGMENT=PAYSEG,$
  FIELDNAME=SOC_SEC_NO,  ALIAS=SSN,      USAGE=A9,      ACTUAL=A9, $
  FIELDNAME=DATE_OF_IN,  ALIAS=INCDATE,  USAGE=I6YMTD,  ACTUAL=A6, $
  FIELDNAME=AMT_OF_INC,  ALIAS=RAISE,    USAGE=D6.2,    ACTUAL=A10,$
  FIELDNAME=PCT_INC,     ALIAS=,         USAGE=D6.2,    ACTUAL=A6, $
  FIELDNAME=NEW_SAL,     ALIAS=CURR_SAL, USAGE=D10.2,   ACTUAL=A11,$
  FIELDNAME=FILL,        ALIAS=,         USAGE=A38,     ACTUAL=A38,$
```

PAYHIST Structure Diagram

```
SECTION 01
           STRUCTURE OF FIX    FILE PAYHIST ON 05/15/03 AT 14.51.59

         PAYSEG
  01     S1
 *************
 *SOC_SEC_NO  **
 *DATE_OF_IN  **
 *AMT_OF_INC  **
 *PCT_INC     **
 *            **
 ***************
  **************
```

COMASTER File

The COMASTER file is used to display the file structure and contents of each segment in a data source. Since COMASTER is used for debugging other Master Files, a corresponding FOCEXEC does not exist for the COMASTER file. Its segments are:

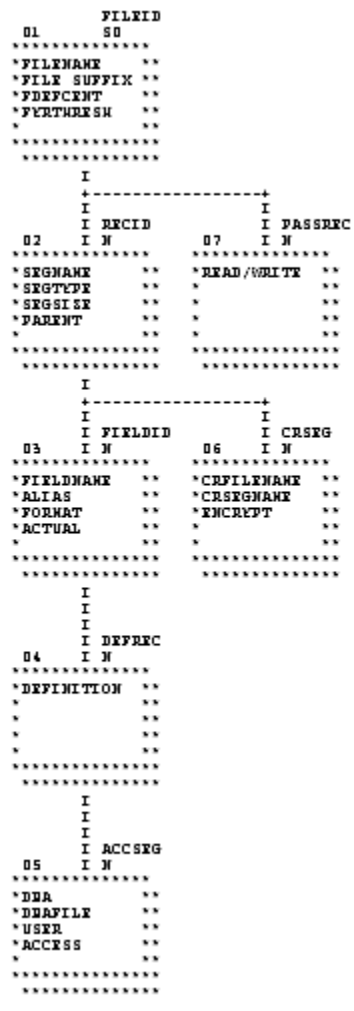
- FILEID, which lists file information.
- RECID, which lists segment information.
- FIELDID, which lists field information.
- DEFREC, which lists a description record.
- PASSREC, which lists read/write access.
- CRSEG, which lists cross-reference information for segments.
- ACCSEG, which lists DBA information.

COMASTER Master File

```
SUFFIX=COM,SEGNAME=FILEID
 FIELDNAME=FILENAME    ,FILE         ,A8 ,      ,$
 FIELDNAME=FILE SUFFIX ,SUFFIX       ,A8 ,      ,$
 FIELDNAME=FDEFCENT    ,FDFC         ,A4 ,      ,$
 FIELDNAME=FYRTHRESH   ,FYRT         ,A2 ,      ,$
SEGNAME=RECID
 FIELDNAME=SEGNAME     ,SEGMENT      ,A8 ,      ,$
 FIELDNAME=SEGTYPE     ,SEGTYPE      ,A4 ,      ,$
 FIELDNAME=SEGSIZE     ,SEGSIZE      ,I4 ,    A4,$
 FIELDNAME=PARENT      ,PARENT       ,A8 ,      ,$
 FIELDNAME=CRKEY       ,VKEY         ,A66,      ,$
SEGNAME=FIELDID
 FIELDNAME=FIELDNAME   ,FIELD        ,A66,      ,$
 FIELDNAME=ALIAS       ,SYNONYM      ,A66,      ,$
 FIELDNAME=FORMAT      ,USAGE        ,A8 ,      ,$
 FIELDNAME=ACTUAL      ,ACTUAL       ,A8 ,      ,$
 FIELDNAME=AUTHORITY   ,AUTHCODE     ,A8 ,      ,$
 FIELDNAME=FIELDTYPE   ,INDEX        ,A8 ,      ,$
 FIELDNAME=TITLE       ,TITLE        ,A64,      ,$
 FIELDNAME=HELPMESSAGE ,MESSAGE      ,A256,     ,$
 FIELDNAME=MISSING     ,MISSING      ,A4 ,      ,$
 FIELDNAME=ACCEPTS     ,ACCEPTABLE   ,A255,     ,$
 FIELDNAME=RESERVED    ,RESERVED     ,A44 ,     ,$
 FIELDNAME=DEFCENT     ,DFC          ,A4 ,      ,$
 FIELDNAME=YRTHRESH    ,YRT          ,A4 ,      ,$
SEGNAME=DEFREC
 FIELDNAME=DEFINITION  ,DESCRIPTION  ,A44,      ,$
SEGNAME=PASSREC,PARENT=FILEID
 FIELDNAME=READ/WRITE  ,RW           ,A32,      ,$
SEGNAME=CRSEG,PARENT=RECID
 FIELDNAME=CRFILENAME  ,CRFILE       ,A8 ,      ,$
 FIELDNAME=CRSEGNAME   ,CRSEGMENT    ,A8 ,      ,$
 FIELDNAME=ENCRYPT     ,ENCRYPT      ,A4 ,      ,$
SEGNAME=ACCSEG,PARENT=DEFREC
 FIELDNAME=DBA         ,DBA          ,A8 ,      ,$
 FIELDNAME=DBAFILE     ,             ,A8 ,      ,$
 FIELDNAME=USER        ,PASS         ,A8 ,      ,$
 FIELDNAME=ACCESS      ,ACCESS       ,A8 ,      ,$
 FIELDNAME=RESTRICT    ,RESTRICT     ,A8 ,      ,$
 FIELDNAME=NAME        ,NAME         ,A66,      ,$
 FIELDNAME=VALUE       ,VALUE        ,A80,      ,$
```

COMASTER Structure Diagram

```
SECTION 01
                STRUCTURE OF EXTERNAL FILE COMASTER ON 05/15/03 AT 14.53.38
```



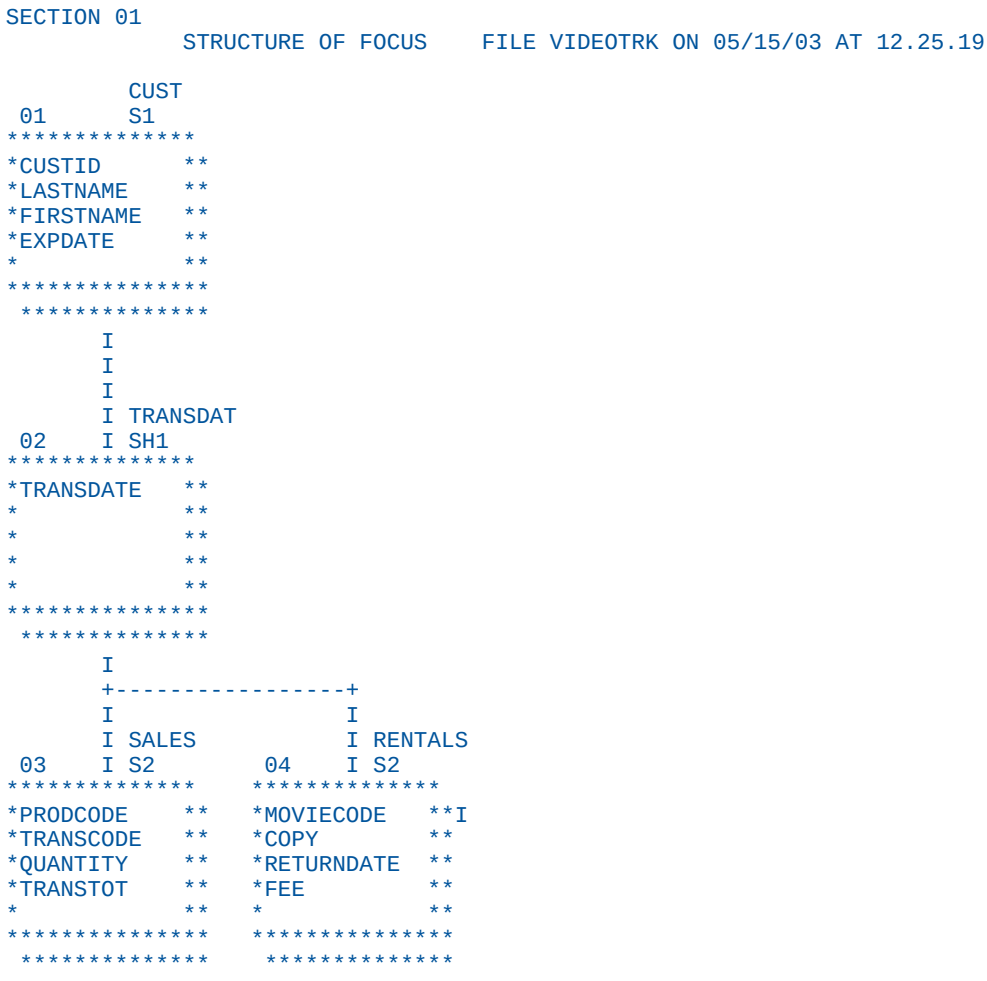
VIDEOTRK, MOVIES, and ITEMS Data Sources

VIDEOTRK contains sample data about customer, rental, and purchase information for a video rental business. It can be joined to the MOVIES or ITEMS data source. VIDEOTRK and MOVIES are used in examples that illustrate the use of the Maintain Data facility.

VIDEOTRK Master File

```
FILENAME=VIDEOTRK, SUFFIX=FOC
 SEGNAME=CUST,     SEGTYPE=S1
  FIELDNAME=CUSTID,     ALIAS=CIN,          FORMAT=A4,    $
  FIELDNAME=LASTNAME,   ALIAS=LN,           FORMAT=A15,   $
  FIELDNAME=FIRSTNAME,  ALIAS=FN,           FORMAT=A10,   $
  FIELDNAME=EXPDATE,    ALIAS=EXDAT,        FORMAT=YMD,   $
  FIELDNAME=PHONE,      ALIAS=TEL,          FORMAT=A10,   $
  FIELDNAME=STREET,     ALIAS=STR,          FORMAT=A20,   $
  FIELDNAME=CITY,       ALIAS=CITY,         FORMAT=A20,   $
  FIELDNAME=STATE,      ALIAS=PROV,         FORMAT=A4,    $
  FIELDNAME=ZIP,        ALIAS=POSTAL_CODE,  FORMAT=A9,    $
SEGNAME=TRANSDAT,  SEGTYPE=SH1,  PARENT=CUST
  FIELDNAME=TRANSDATE,  ALIAS=OUTDATE,      FORMAT=YMD,   $
SEGNAME=SALES,     SEGTYPE=S2,   PARENT=TRANSDAT
  FIELDNAME=PRODCODE,   ALIAS=PCOD,         FORMAT=A6,    $
  FIELDNAME=TRANSCODE,  ALIAS=TCOD,         FORMAT=I3,    $
  FIELDNAME=QUANTITY,   ALIAS=NO,           FORMAT=I3S,   $
  FIELDNAME=TRANSTOT,   ALIAS=TTOT,         FORMAT=F7.2S, $
SEGNAME=RENTALS,   SEGTYPE=S2,   PARENT=TRANSDAT
  FIELDNAME=MOVIECODE,  ALIAS=MCOD,         FORMAT=A6, INDEX=I, $
  FIELDNAME=COPY,       ALIAS=COPY,         FORMAT=I2,    $
  FIELDNAME=RETURNDATE, ALIAS=INDATE,       FORMAT=YMD,   $
  FIELDNAME=FEE,        ALIAS=FEE,          FORMAT=F5.2S, $
```

VIDEOTRK Structure Diagram



MOVIES Master File

```
FILENAME=MOVIES,     SUFFIX=FOC
 SEGNAME=MOVINFO,    SEGTYPE=S1
  FIELDNAME=MOVIECODE,    ALIAS=MCOD,  FORMAT=A6, INDEX=I, $
  FIELDNAME=TITLE,        ALIAS=MTL,   FORMAT=A39,  $
  FIELDNAME=CATEGORY,     ALIAS=CLASS, FORMAT=A8,   $
  FIELDNAME=DIRECTOR,     ALIAS=DIR,   FORMAT=A17,  $
  FIELDNAME=RATING,       ALIAS=RTG,   FORMAT=A4,   $
  FIELDNAME=RELDATE,      ALIAS=RDAT,  FORMAT=YMD,  $
  FIELDNAME=WHOLESALEPR, ALIAS=WPRC,  FORMAT=F6.2, $
  FIELDNAME=LISTPR,       ALIAS=LPRC,  FORMAT=F6.2, $
  FIELDNAME=COPIES,       ALIAS=NOC,   FORMAT=I3,   $
```

MOVIES Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE MOVIES   ON 05/15/03 AT 12.26.05

        MOVINFO
 01     S1
**************
*MOVIECODE   **I
*TITLE       **
*CATEGORY    **
*DIRECTOR    **
*            **
***************
 **************
```

ITEMS Master File

```
FILENAME=ITEMS,    SUFFIX=FOC
 SEGNAME=ITMINFO, SEGTYPE=S1
  FIELDNAME=PRODCODE,  ALIAS=PCOD,  FORMAT=A6,  INDEX=I, $
  FIELDNAME=PRODNAME,  ALIAS=PROD,  FORMAT=A20,          $
  FIELDNAME=OURCOST,   ALIAS=WCOST, FORMAT=F6.2,         $
  FIELDNAME=RETAILPR,  ALIAS=PRICE, FORMAT=F6.2,         $
  FIELDNAME=ON_HAND,   ALIAS=NUM,   FORMAT=I5,           $
```

ITEMS Structure Diagram

```
SECTION 01
            STRUCTURE OF FOCUS    FILE ITEMS    ON 05/15/03 AT 12.26.05

         ITMINFO
 01      S1
**************
*PRODCODE    **I
*PRODNAME    **
*OURCOST     **
*RETAILPR    **
*            **
***************
 **************
```

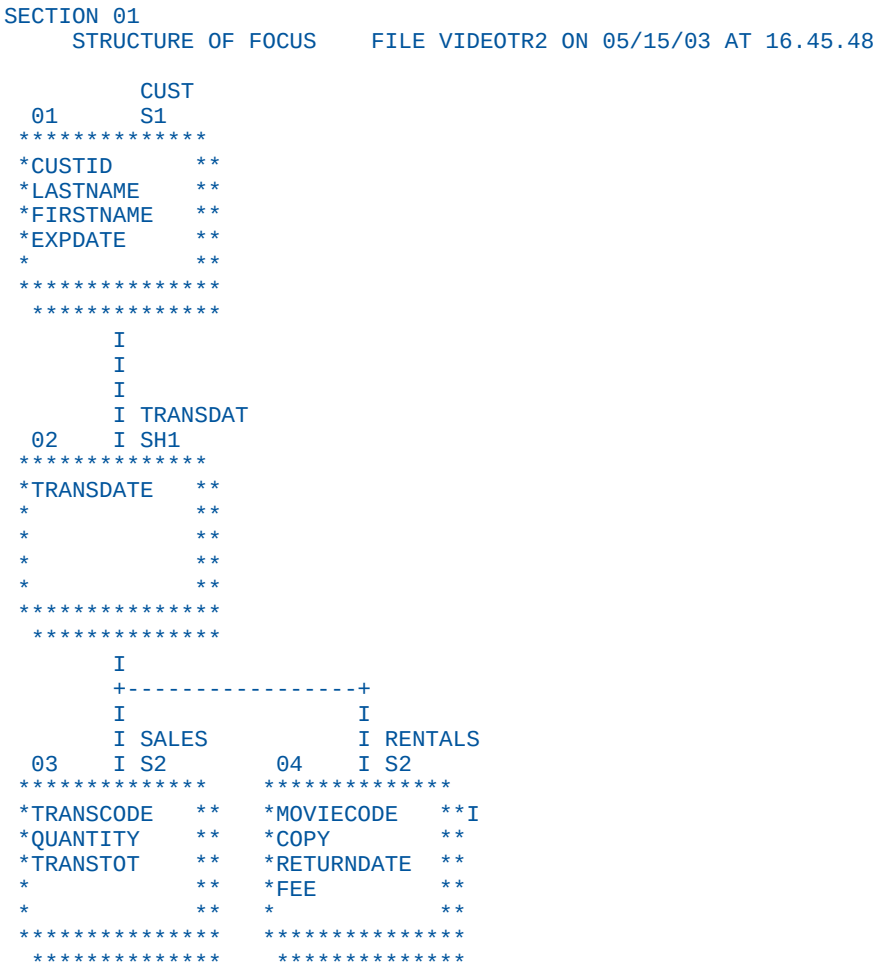
VIDEOTR2 Data Source

VIDEOTR2 contains sample data about customer, rental, and purchase information for a video rental business. It consists of four segments.

VIDEOTR2 Master File

```
FILENAME=VIDEOTR2, SUFFIX=FOC
 SEGNAME=CUST,     SEGTYPE=S1
  FIELDNAME=CUSTID,       ALIAS=CIN,          FORMAT=A4,          $
  FIELDNAME=LASTNAME,     ALIAS=LN,           FORMAT=A15,         $
  FIELDNAME=FIRSTNAME,    ALIAS=FN,           FORMAT=A10,         $
  FIELDNAME=EXPDATE,      ALIAS=EXDAT,        FORMAT=YMD,         $
  FIELDNAME=PHONE,        ALIAS=TEL,          FORMAT=A10,         $
  FIELDNAME=STREET,       ALIAS=STR,          FORMAT=A20,         $
  FIELDNAME=CITY,         ALIAS=CITY,         FORMAT=A20,         $
  FIELDNAME=STATE,        ALIAS=PROV,         FORMAT=A4,          $
  FIELDNAME=ZIP,          ALIAS=POSTAL_CODE,  FORMAT=A9,          $
  FIELDNAME=EMAIL,        ALIAS=EMAIL,        FORMAT=A18,         $
 SEGNAME=TRANSDAT, SEGTYPE=SH1,  PARENT=CUST
  FIELDNAME=TRANSDATE,    ALIAS=OUTDATE,      FORMAT=HYYMDI,      $
 SEGNAME=SALES,    SEGTYPE=S2,   PARENT=TRANSDAT
  FIELDNAME=TRANSCODE,    ALIAS=TCOD,         FORMAT=I3,          $
  FIELDNAME=QUANTITY,     ALIAS=NO,           FORMAT=I3S,         $
  FIELDNAME=TRANSTOT,     ALIAS=TTOT,         FORMAT=F7.2S,       $
 SEGNAME=RENTALS,  SEGTYPE=S2,   PARENT=TRANSDAT
  FIELDNAME=MOVIECODE,    ALIAS=MCOD,         FORMAT=A6, INDEX=I, $
  FIELDNAME=COPY,         ALIAS=COPY,         FORMAT=I2,          $
  FIELDNAME=RETURNDATE,   ALIAS=INDATE,       FORMAT=YMD,         $
  FIELDNAME=FEE,          ALIAS=FEE,          FORMAT=F5.2S,       $
```

VIDEOTR2 Structure Diagram



Gotham Grinds Data Sources

Gotham Grinds is a group of data sources that contain sample data about a specialty items company.

- GGDEMOG contains demographic information about the customers of Gotham Grinds, a company that sells specialty items like coffee, gourmet snacks, and gifts. It consists of one segment, DEMOG01.
- GGORDER contains order information for Gotham Grinds. It consists of two segments, ORDER01 and ORDER02.

- GGPRODS contains product information for Gotham Grinds. It consists of one segment, PRODS01.
- GGSALES contains sales information for Gotham Grinds. It consists of one segment, SALES01.
- GGSTORES contains information for each of Gotham Grinds 12 stores in the United States. It consists of one segment, STORES01.

GGDEMOG Master File

```
FILENAME=GGDEMOG, SUFFIX=FOC
 SEGNAME=DEMOG01, SEGTYPE=S1
  FIELD=ST,       ALIAS=E02, FORMAT=A02, INDEX=I,TITLE='State',
   DESC='State',$
  FIELD=HH,       ALIAS=E03, FORMAT=I09, TITLE='Number of Households',
   DESC='Number of Households',$
  FIELD=AVGHHSZ98,ALIAS=E04, FORMAT=I09, TITLE='Average Household Size',
   DESC='Average Household Size',$
  FIELD=MEDHHI98, ALIAS=E05, FORMAT=I09, TITLE='Median Household Income',
   DESC='Median Household Income',$
  FIELD=AVGHHI98, ALIAS=E06, FORMAT=I09, TITLE='Average Household Income',
   DESC='Average Household Income',$
  FIELD=MALEPOP98,ALIAS=E07, FORMAT=I09, TITLE='Male Population',
   DESC='Male Population',$
  FIELD=FEMPOP98, ALIAS=E08, FORMAT=I09, TITLE='Female Population',
   DESC='Female Population',$
  FIELD=P15TO1998,ALIAS=E09, FORMAT=I09, TITLE='15 to 19',
   DESC='Population 15 to 19 years old',$
  FIELD=P20TO2998,ALIAS=E10, FORMAT=I09, TITLE='20 to 29',
   DESC='Population 20 to 29 years old',$
  FIELD=P30TO4998,ALIAS=E11, FORMAT=I09, TITLE='30 to 49',
   DESC='Population 30 to 49 years old',$
  FIELD=P50TO6498,ALIAS=E12, FORMAT=I09, TITLE='50 to 64',
   DESC='Population 50 to 64 years old',$
  FIELD=P65OVR98, ALIAS=E13, FORMAT=I09, TITLE='65 and over',
   DESC='Population 65 and over',$
```

GGDEMOG Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE GGDEMOG  ON 05/15/03 AT 12.26.05

         GGDEMOG
 01      S1
**************
*ST          **I
*HH          **
*AVGHHSZ98   **
*MEDHHI98    **
*            **
***************
 **************
```

GGORDER Master File

```
FILENAME=GGORDER, SUFFIX=FOC,$
 SEGNAME=ORDER01, SEGTYPE=S1,$
  FIELD=ORDER_NUMBER, ALIAS=ORDNO1,   FORMAT=I6,  TITLE='Order,Number',
   DESC='Order Identification Number',$
  FIELD=ORDER_DATE,   ALIAS=DATE,     FORMAT=MDY, TITLE='Order,Date',
   DESC='Date order was placed',$
  FIELD=STORE_CODE,   ALIAS=STCD,     FORMAT=A5,  TITLE='Store,Code',
   DESC='Store Identification Code (for order)',$
  FIELD=PRODUCT_CODE, ALIAS=PCD,      FORMAT=A4,  TITLE='Product,Code',
   DESC='Product Identification Code (for order)',$
  FIELD=QUANTITY,     ALIAS=ORDUNITS, FORMAT=I8,  TITLE='Ordered,Units',
   DESC='Quantity Ordered',$
SEGNAME=ORDER02, SEGTYPE=KU, PARENT=ORDER01, CRFILE=GGPRODS, CRKEY=PCD,
CRSEG=PRODS01  ,$
```

GGORDER Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE GGORDER  ON 05/15/03 AT 16.45.48

          GGORDER
  01      S1
 **************
 *ORDER_NUMBER**
 *ORDER_DATE  **
 *STORE_CODE  **
 *PRODUCT_CODE**
 *            **
 ***************
  **************
        I
        I
        I
        I ORDER02
  02    I KU
 ..............
 :PRODUCT_ID  :K
 :PRODUCT_DESC:
 :VENDOR_CODE :
 :VENDOR_NAME :
 :            :
 :............:
```

GGPRODS Master File

```
FILENAME=GGPRODS, SUFFIX=FOC
 SEGNAME=PRODS01, SEGTYPE=S1
  FIELD=PRODUCT_ID, ALIAS=PCD, FORMAT=A4, INDEX=I, TITLE='Product,Code',
   DESC='Product Identification Code',$
  FIELD=PRODUCT_DESCRIPTION, ALIAS=PRODUCT, FORMAT=A16, TITLE='Product',
   DESC='Product Name',$
  FIELD=VENDOR_CODE, ALIAS=VCD, FORMAT=A4, INDEX=I, TITLE='Vendor ID',
   DESC='Vendor Identification Code',$
  FIELD=VENDOR_NAME, ALIAS=VENDOR, FORMAT=A23, TITLE='Vendor Name',
   DESC='Vendor Name',$
  FIELD=PACKAGE_TYPE, ALIAS=PACK, FORMAT=A7, TITLE='Package',
   DESC='Packaging Style',$
  FIELD=SIZE, ALIAS=SZ, FORMAT=I2, TITLE='Size',
   DESC='Package Size',$
  FIELD=UNIT_PRICE, ALIAS=UNITPR, FORMAT=D7.2, TITLE='Unit,Price',
   DESC='Price for one unit',$
```

GGPRODS Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE GGPRODS  ON 05/15/03 AT 12.26.05

         GGPRODS
 01      S1
*************
*PRODUCT_ID  **I
*PRODUCT_DESC**I
*VENDOR_CODE **
*VENDOR_NAME **
*             **
**************
 *************
```

GGSALES Master File

```
FILENAME=GGSALES, SUFFIX=FOC
 SEGNAME=SALES01, SEGTYPE=S1
  FIELD=SEQ_NO, ALIAS=SEQ, FORMAT=I5, TITLE='Sequence#',
   DESC='Sequence number in database',$
  FIELD=CATEGORY, ALIAS=E02, FORMAT=A11, INDEX=I, TITLE='Category',
   DESC='Product category',$
  FIELD=PCD, ALIAS=E03, FORMAT=A04, INDEX=I, TITLE='Product ID',
   DESC='Product Identification code (for sale)',$
  FIELD=PRODUCT, ALIAS=E04, FORMAT=A16, TITLE='Product',
   DESC='Product name',$
  FIELD=REGION, ALIAS=E05, FORMAT=A11, INDEX=I, TITLE='Region',
   DESC='Region code',$
  FIELD=ST, ALIAS=E06, FORMAT=A02, INDEX=I, TITLE='State',
   DESC='State',$
  FIELD=CITY, ALIAS=E07, FORMAT=A20, TITLE='City',
   DESC='City',$
  FIELD=STCD, ALIAS=E08, FORMAT=A05, INDEX=I, TITLE='Store ID',
   DESC='Store identification code (for sale)',$
  FIELD=DATE, ALIAS=E09, FORMAT=I8YYMD, TITLE='Date',
   DESC='Date of sales report',$
  FIELD=UNITS, ALIAS=E10, FORMAT=I08, TITLE='Unit Sales',
   DESC='Number of units sold',$
  FIELD=DOLLARS, ALIAS=E11, FORMAT=I08, TITLE='Dollar Sales',
   DESC='Total dollar amount of reported sales',$
  FIELD=BUDUNITS, ALIAS=E12, FORMAT=I08, TITLE='Budget Units',
   DESC='Number of units budgeted',$
  FIELD=BUDDOLLARS, ALIAS=E13, FORMAT=I08, TITLE='Budget Dollars',
   DESC='Total sales quota in dollars',$
```

GGSALES Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE GGSALES  ON 05/15/03 AT 12.26.05

         GGSALES
 01      S1
**************
*SEQ_NO       **
*CATEGORY     **I
*PCD          **I
*PRODUCT      **I
*             **
***************
 **************
```

GGSTORES Master File

```
FILENAME=GGSTORES, SUFFIX=FOC
 SEGNAME=STORES01, SEGTYPE=S1
  FIELD=STORE_CODE, ALIAS=E02, FORMAT=A05, INDEX=I, TITLE='Store ID',
   DESC='Franchisee ID Code',$
  FIELD=STORE_NAME, ALIAS=E03, FORMAT=A23, TITLE='Store Name',
   DESC='Store Name',$
  FIELD=ADDRESS1, ALIAS=E04, FORMAT=A19, TITLE='Contact',
   DESC='Franchisee Owner',$
  FIELD=ADDRESS2, ALIAS=E05, FORMAT=A31, TITLE='Address',
   DESC='Street Address',$
  FIELD=CITY, ALIAS=E06, FORMAT=A22, TITLE='City',
   DESC='City',$
  FIELD=STATE, ALIAS=E07, FORMAT=A02, INDEX=I, TITLE='State',
   DESC='State',$
  FIELD=ZIP, ALIAS=E08, FORMAT=A06, TITLE='Zip Code',
   DESC='Postal Code',$
```

GGSTORES Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE GGSTORES ON 05/15/03 AT 12.26.05

         GGSTORES
 01      S1
**************
*STORE_CODE   **I
*STORE_NAME   **
*ADDRESS1     **
*ADDRESS2     **
*             **
***************
 **************
```

Century Corp Data Sources

Century Corp is a consumer electronics manufacturer that distributes products through retailers around the world. Century Corp has thousands of employees in plants, warehouses, and offices worldwide. Their mission is to provide quality products and services to their customers.

Century Corp is a group of data sources that contain financial, human resources, inventory, and order information. The last three data sources are designed to be used with chart of accounts data.

- CENTCOMP Master File contains location information for stores. It consists of one segment, COMPINFO.
- CENTFIN Master File contains financial information. It consists of one segment, ROOT_SEG.
- CENTHR Master File contains human resources information. It consists of one segment, EMPSEG.
- CENTINV Master File contains inventory information. It consists of one segment, INVINFO.
- CENTORD Master File contains order information. It consists of four segments, OINFO, STOSEG, PINFO, and INVSEG.
- CENTQA Master File contains problem information. It consists of three segments, PROD_SEG, INVSEG, and PROB_SEG.
- CENTGL Master File contains a chart of accounts hierarchy. The field GL_ACCOUNT_PARENT is the parent field in the hierarchy. The field GL_ACCOUNT is the hierarchy field. The field GL_ACCOUNT_CAPTION can be used as the descriptive caption for the hierarchy field.
- CENTSYSF Master File contains detail-level financial data. CENTSYSF uses a different account line system (SYS_ACCOUNT), which can be joined to the SYS_ACCOUNT field in CENTGL. Data uses "natural" signs (expenses are positive, revenue negative).
- CENTSTMT Master File contains detail-level financial data and a cross-reference to the CENTGL data source.

CENTCOMP Master File

```
FILE=CENTCOMP, SUFFIX=FOC, FDFC=19, FYRT=00
  SEGNAME=COMPINFO, SEGTYPE=S1, $
  FIELD=STORE_CODE, ALIAS=SNUM, FORMAT=A6, INDEX=I,
   TITLE='Store Id#:',
   DESCRIPTION='Store Id#', $
  FIELD=STORENAME, ALIAS=SNAME, FORMAT=A20,
   WITHIN=STATE,
   TITLE='Store,Name:',
   DESCRIPTION='Store Name', $
  FIELD=STATE, ALIAS=STATE, FORMAT=A2,
   WITHIN=PLANT,
   TITLE='State:',
   DESCRIPTION=State, $
  DEFINE REGION/A5=DECODE STATE ('AL' 'SOUTH' 'AK' 'WEST' 'AR' 'SOUTH'
  'AZ' 'WEST' 'CA' 'WEST' 'CO' 'WEST' 'CT' 'EAST'
  'DE' 'EAST' 'DC' 'EAST' 'FL' 'SOUTH' 'GA' 'SOUTH' 'HI' 'WEST'
  'ID' 'WEST' 'IL' 'NORTH' 'IN' 'NORTH' 'IA' 'NORTH'
  'KS' 'NORTH' 'KY' 'SOUTH' 'LA' 'SOUTH' 'ME' 'EAST' 'MD' 'EAST'
  'MA' 'EAST' 'MI' 'NORTH' 'MN' 'NORTH' 'MS' 'SOUTH' 'MT' 'WEST'
  'MO' 'SOUTH' 'NE' 'WEST'  'NV' 'WEST' 'NH' 'EAST' 'NJ' 'EAST'
  'NM' 'WEST' 'NY' 'EAST' 'NC' 'SOUTH' 'ND' 'NORTH' 'OH' 'NORTH'
  'OK' 'SOUTH' 'OR' 'WEST' 'PA' 'EAST' 'RI' 'EAST' 'SC' 'SOUTH'
  'SD' 'NORTH' 'TN' 'SOUTH' 'TX' 'SOUTH' 'UT' 'WEST' 'VT' 'EAST'
  'VA' 'SOUTH' 'WA' 'WEST' 'WV' 'SOUTH' 'WI' 'NORTH' 'WY' 'WEST'
  'NA' 'NORTH' 'ON' 'NORTH' ELSE ' ');,
   TITLE='Region:',
   DESCRIPTION=Region, $
```

CENTCOMP Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTCOMP ON 05/15/03 AT 10.20.49

          COMPINFO
  01      S1
 **************
 *STORE_CODE   **I
 *STORENAME    **
 *STATE        **
 *             **
 *             **
 ***************
  **************
```

CENTFIN Master File

```
FILE=CENTFIN, SUFFIX=FOC, FDFC=19, FYRT=00
  SEGNAME=ROOT_SEG, SEGTYPE=S4, $
  FIELD=YEAR, ALIAS=YEAR, FORMAT=YY,
   WITHIN='*Time Period', $
  FIELD=QUARTER, ALIAS=QTR, FORMAT=Q,
   WITHIN=YEAR,
   TITLE=Quarter,
   DESCRIPTION=Quarter, $
  FIELD=MONTH, ALIAS=MONTH, FORMAT=M,
   TITLE=Month,
   DESCRIPTION=Month, $
  FIELD=ITEM, ALIAS=ITEM, FORMAT=A20,
   TITLE=Item,
   DESCRIPTION=Item, $
  FIELD=VALUE, ALIAS=VALUE, FORMAT=D12.2,
   TITLE=Value,
   DESCRIPTION=Value, $
  DEFINE ITYPE/A12=IF EDIT(ITEM,'9$$$$$$$$$$$$$$$$$$$') EQ 'E'
   THEN 'Expense' ELSE IF EDIT(ITEM,'9$$$$$$$$$$$$$$$$$$$') EQ 'R'
   THEN 'Revenue' ELSE 'Asset';,
   TITLE=Type,
   DESCRIPTION='Type of Financial Line Item',$
  DEFINE MOTEXT/MT=MONTH;,$
```

CENTFIN Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTFIN  ON 05/15/03 AT 10.25.52

          ROOT_SEG
  01      S4
 **************
 *YEAR         **
 *QUARTER      **
 *MONTH        **
 *ITEM         **
 *             **
 ***************
  **************
```

CENTHR Master File

```
FILE=CENTHR, SUFFIX=FOC
  SEGNAME=EMPSEG, SEGTYPE=S1, $
  FIELD=ID_NUM, ALIAS=ID#, FORMAT=I9,
   TITLE='Employee,ID#',
   DESCRIPTION='Employee Identification Number', $
  FIELD=LNAME, ALIAS=LN, FORMAT=A14,
   TITLE='Last,Name',
   DESCRIPTION='Employee Last Name', $
  FIELD=FNAME, ALIAS=FN, FORMAT=A12,
   TITLE='First,Name',
   DESCRIPTION='Employee First Name', $
  FIELD=PLANT, ALIAS=PLT, FORMAT=A3,
   TITLE='Plant,Location',
   DESCRIPTION='Location of the manufacturing plant',
   WITHIN='*Location', $
  FIELD=START_DATE, ALIAS=SDATE, FORMAT=YYMD,
   TITLE='Starting,Date',
   DESCRIPTION='Date of employment',$
  FIELD=TERM_DATE, ALIAS=TERM_DATE, FORMAT=YYMD,
   TITLE='Termination,Date',
   DESCRIPTION='Termination Date', $
  FIELD=STATUS, ALIAS=STATUS, FORMAT=A10,
   TITLE='Current,Status',
   DESCRIPTION='Job Status', $
  FIELD=POSITION, ALIAS=JOB, FORMAT=A2,
   TITLE=Position,
   DESCRIPTION='Job Position',  $
  FIELD=PAYSCALE, ALIAS=PAYLEVEL, FORMAT=I2,
   TITLE='Pay,Level',
   DESCRIPTION='Pay Level',
   WITHIN='*Wages',$
  DEFINE POSITION_DESC/A17=IF POSITION EQ 'BM' THEN
   'Plant Manager' ELSE
   IF POSITION EQ 'MR' THEN 'Line Worker' ELSE
   IF POSITION EQ 'TM' THEN 'Line Manager' ELSE
   'Technician';
   TITLE='Position,Description',
   DESCRIPTION='Position Description',
   WITHIN='PLANT',$
  DEFINE BYEAR/YY=START_DATE;
   TITLE='Beginning,Year',
   DESCRIPTION='Beginning Year',
   WITHIN='*Starting Time Period',$
```

```
DEFINE BQUARTER/Q=START_DATE;
 TITLE='Beginning,Quarter',
 DESCRIPTION='Beginning Quarter',
 WITHIN='BYEAR',
DEFINE BMONTH/M=START_DATE;
 TITLE='Beginning,Month',
 DESCRIPTION='Beginning Month',
 WITHIN='BQUARTER',$
DEFINE EYEAR/YY=TERM_DATE;
 TITLE='Ending,Year',
 DESCRIPTION='Ending Year',
 WITHIN='*Termination Time Period',$
DEFINE EQUARTER/Q=TERM_DATE;
 TITLE='Ending,Quarter',
 DESCRIPTION='Ending Quarter',
 WITHIN='EYEAR',$
DEFINE EMONTH/M=TERM_DATE;
 TITLE='Ending,Month',
 DESCRIPTION='Ending Month',
 WITHIN='EQUARTER',$
DEFINE RESIGN_COUNT/I3=IF STATUS EQ 'RESIGNED' THEN 1
 ELSE 0;
 TITLE='Resigned,Count',
 DESCRIPTION='Resigned Count',$
DEFINE FIRE_COUNT/I3=IF STATUS EQ 'TERMINAT' THEN 1
 ELSE 0;
 TITLE='Terminated,Count',
 DESCRIPTION='Terminated Count',$
DEFINE DECLINE_COUNT/I3=IF STATUS EQ 'DECLINED' THEN 1
 ELSE 0;
 TITLE='Declined,Count',
 DESCRIPTION='Declined Count',$
DEFINE EMP_COUNT/I3=IF STATUS EQ 'EMPLOYED' THEN 1
 ELSE 0;
 TITLE='Employed,Count',
 DESCRIPTION='Employed Count',$
DEFINE PEND_COUNT/I3=IF STATUS EQ 'PENDING' THEN 1
 ELSE 0;
 TITLE='Pending,Count',
 DESCRIPTION='Pending Count',$
DEFINE REJECT_COUNT/I3=IF STATUS EQ 'REJECTED' THEN 1
 ELSE 0;
 TITLE='Rejected,Count',
 DESCRIPTION='Rejected Count',$
DEFINE FULLNAME/A28=LNAME||', '|FNAME;
 TITLE='Full Name',
 DESCRIPTION='Full Name: Last, First', WITHIN='POSITION_DESC',$
```

```
 DEFINE SALARY/D12.2=IF BMONTH LT 4 THEN PAYLEVEL * 12321
  ELSE IF BMONTH GE 4 AND BMONTH LT 8 THEN PAYLEVEL * 13827
  ELSE PAYLEVEL * 14400;,
  TITLE='Salary',
  DESCRIPTION='Salary',$
 DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'
  LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'
  ELSE 'n/a');$
```

CENTHR Structure Diagram

```
SECTION 01
     STRUCTURE OF FOCUS    FILE CENTHR   ON 05/15/03 AT 10.40.34

          EMPSEG
  01      S1
 **************
 *ID_NUM      **
 *LNAME       **
 *FNAME       **
 *PLANT       **
 *            **
 ***************
  **************
```

CENTINV Master File

```
FILE=CENTINV, SUFFIX=FOC, FDFC=19, FYRT=00
 SEGNAME=INVINFO, SEGTYPE=S1, $
  FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4, INDEX=I,
   TITLE='Product,Number:',
   DESCRIPTION='Product Number', $
  FIELD=PRODNAME, ALIAS=PNAME, FORMAT=A30,
   WITHIN=PRODCAT,
   TITLE='Product,Name:',
   DESCRIPTION='Product Name', $
  FIELD=QTY_IN_STOCK, ALIAS=QIS, FORMAT=I7,
   TITLE='Quantity,In Stock:',
   DESCRIPTION='Quantity In Stock', $
  FIELD=PRICE, ALIAS=RETAIL, FORMAT=D10.2,
   TITLE='Price:',
   DESCRIPTION=Price, $
  FIELD=COST, ALIAS=OUR_COST, FORMAT=D10.2,
   TITLE='Our,Cost:',
   DESCRIPTION='Our Cost:', $
  DEFINE PRODCAT/A22 = IF PRODNAME CONTAINS 'LCD'
   THEN 'VCRs' ELSE IF PRODNAME
   CONTAINS 'DVD' THEN 'DVD' ELSE IF PRODNAME CONTAINS 'Camcor'
   THEN 'Camcorders'
   ELSE IF PRODNAME CONTAINS 'Camera' THEN 'Cameras' ELSE IF PRODNAME
   CONTAINS 'CD' THEN 'CD Players'
   ELSE IF PRODNAME CONTAINS 'Tape' THEN 'Digital Tape Recorders'
   ELSE IF PRODNAME CONTAINS 'Combo' THEN 'Combo Players'
   ELSE 'PDA Devices'; WITHIN=PRODTYPE, TITLE='Product Category:' ,$
  DEFINE PRODTYPE/A19 = IF PRODNAME CONTAINS 'Digital' OR 'DVD' OR 'QX'
   THEN 'Digital' ELSE 'Analog';,WITHIN='*Product Dimension',
   TITLE='Product Type:',$
```

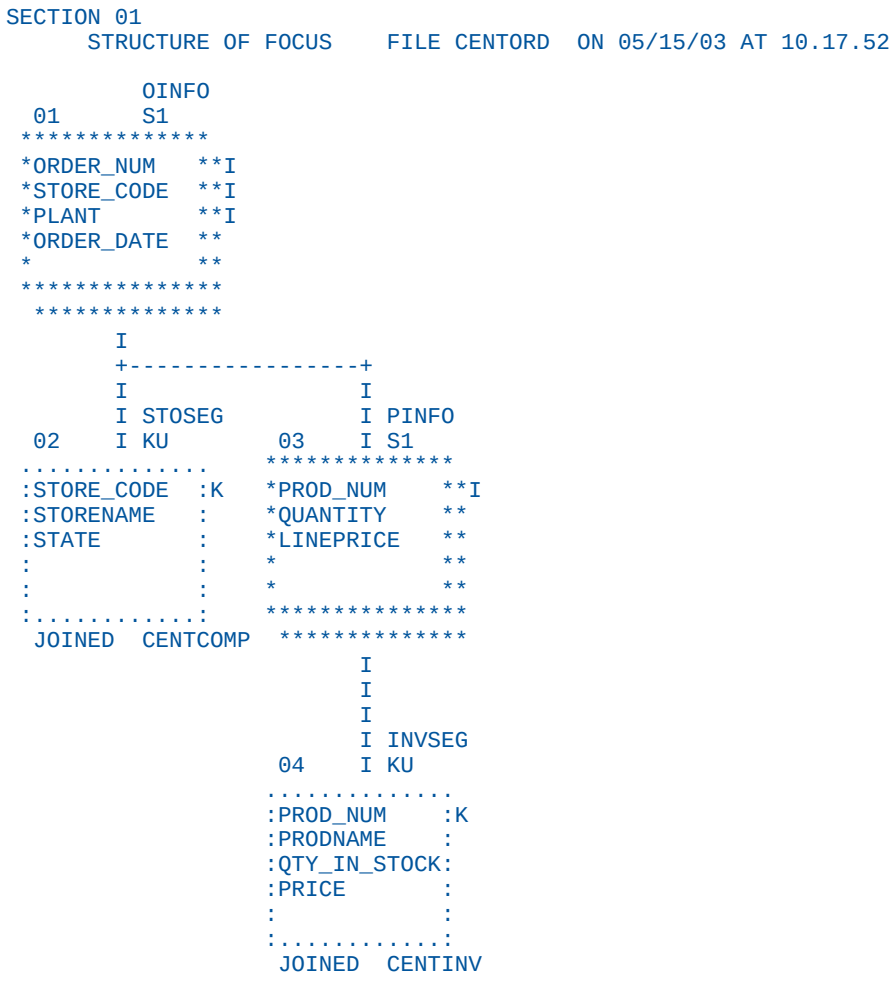
CENTINV Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTINV  ON 05/15/03 AT 10.43.35

          INVINFO
  01      S1
 **************
 *PROD_NUM    **I
 *PRODNAME    **
 *QTY_IN_STOCK**
 *PRICE       **
 *            **
 ***************
  ***************
```

CENTORD Master File

```
FILE=CENTORD, SUFFIX=FOC
 SEGNAME=OINFO, SEGTYPE=S1, $
  FIELD=ORDER_NUM, ALIAS=ONUM, FORMAT=A5, INDEX=I,
   TITLE='Order,Number:',
   DESCRIPTION='Order Number', $
  FIELD=ORDER_DATE, ALIAS=ODATE, FORMAT=YYMD,
   TITLE='Date,Of Order:',
   DESCRIPTION='Date Of Order', $
  FIELD=STORE_CODE, ALIAS=SNUM, FORMAT=A6, INDEX=I,
   TITLE='Company ID#:',
   DESCRIPTION='Company ID#', $
  FIELD=PLANT, ALIAS=PLNT, FORMAT=A3, INDEX=I,
   TITLE='Manufacturing,Plant',
   DESCRIPTION='Location Of Manufacturing Plant',
   WITHIN='*Location',$
  DEFINE YEAR/YY=ORDER_DATE;,
   WITHIN='*Time Period',$
  DEFINE QUARTER/Q=ORDER_DATE;,
   WITHIN='YEAR',$
  DEFINE MONTH/M=ORDER_DATE;,
   WITHIN='QUARTER',$
 SEGNAME=PINFO, SEGTYPE=S1, PARENT=OINFO, $
  FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4,INDEX=I,
   TITLE='Product,Number#:',
   DESCRIPTION='Product Number#', $
  FIELD=QUANTITY, ALIAS=QTY, FORMAT=I8C,
   TITLE='Quantity:',
   DESCRIPTION=Quantity, $
  FIELD=LINEPRICE, ALIAS=LINETOTAL, FORMAT=D12.2MC,
   TITLE='Line,Total',
   DESCRIPTION='Line Total', $
  DEFINE LINE_COGS/D12.2=QUANTITY*COST;,
   TITLE='Line,Cost Of,Goods Sold',
   DESCRIPTION='Line cost of goods sold', $
  DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'
   LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'
   ELSE 'n/a');
 SEGNAME=INVSEG, SEGTYPE=DKU, PARENT=PINFO, CRFILE=CENTINV,
  CRKEY=PROD_NUM,   CRSEG=INVINFO,$
 SEGNAME=STOSEG, SEGTYPE=DKU, PARENT=OINFO, CRFILE=CENTCOMP,
  CRKEY=STORE_CODE, CRSEG=COMPINFO,$
```

CENTORD Structure Diagram



CENTQA Master File

```
FILE=CENTQA, SUFFIX=FOC, FDFC=19, FYRT=00
 SEGNAME=PROD_SEG, SEGTYPE=S1, $
  FIELD=PROD_NUM, ALIAS=PNUM, FORMAT=A4, INDEX=I,
   TITLE='Product,Number',
   DESCRIPTION='Product Number', $
 SEGNAME=PROB_SEG, PARENT=PROD_SEG, SEGTYPE=S1, $
  FIELD=PROBNUM, ALIAS=PROBNO, FORMAT=I5,
   TITLE='Problem,Number',
   DESCRIPTION='Problem Number',
   WITHIN=PLANT,$
  FIELD=PLANT, ALIAS=PLT, FORMAT=A3, INDEX=I,
   TITLE=Plant,
   DESCRIPTION=Plant,
   WITHIN=PROBLEM_LOCATION,$
  FIELD=PROBLEM_DATE, ALIAS=PDATE, FORMAT=YYMD,
   TITLE='Date,Problem,Reported',
   DESCRIPTION='Date Problem Was Reported', $
  FIELD=PROBLEM_CATEGORY, ALIAS=PROBCAT, FORMAT=A20, $
   TITLE='Problem,Category',
   DESCRIPTION='Problem Category',
   WITHIN=*Problem,$
  FIELD=PROBLEM_LOCATION, ALIAS=PROBLOC, FORMAT=A10,
   TITLE='Location,Problem,Occurred',
   DESCRIPTION='Location Where Problem Occurred',
   WITHIN=PROBLEM_CATEGORY,$
  DEFINE PROB_YEAR/YY=PROBLEM_DATE;,
   TITLE='Year,Problem,Occurred',
   DESCRIPTION='Year Problem Occurred',
   WITHIN=*Time Period,$
  DEFINE PROB_QUARTER/Q=PROBLEM_DATE;
   TITLE='Quarter,Problem,Occurred',
   DESCRIPTION='Quarter Problem Occurred',
   WITHIN=PROB_YEAR,$
  DEFINE PROB_MONTH/M=PROBLEM_DATE;
   TITLE='Month,Problem,Occurred',
   DESCRIPTION='Month Problem Occurred',
   WITHIN=PROB_QUARTER,$
  DEFINE PROBLEM_OCCUR/I5 WITH PROBNUM=1;,
   TITLE='Problem,Occurrence'
   DESCRIPTION='# of times a problem occurs',$
  DEFINE PLANTLNG/A11=DECODE PLANT (BOS 'Boston' DAL 'Dallas'
   LA 'Los Angeles' ORL 'Orlando' SEA 'Seattle' STL 'St Louis'
   ELSE 'n/a');$
 SEGNAME=INVSEG, SEGTYPE=DKU, PARENT=PROD_SEG, CRFILE=CENTINV,
  CRKEY=PROD_NUM, CRSEG=INVINFO,$
```

CENTQA Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTQA   ON 05/15/03 AT 10.46.43

          PROD_SEG
 01       S1
 **************
 *PROD_NUM    **I
 *            **
 *            **
 *            **
 *            **
 ***************
  **************
        I
        +-----------------+
        I                 I
        I INVSEG          I PROB_SEG
 02     I KU         03    I S1
 ..............    **************
 :PROD_NUM    :K   *PROBNUM     **
 :PRODNAME    :    *PLANT       **I
 :QTY_IN_STOCK:    *PROBLEM_DATE**
 :PRICE       :    *PROBLEM_CAT>**
 :            :    *             **
 :............:    **************
  JOINED  CENTINV   **************
```

CENTGL Master File

```
FILE=CENTGL ,SUFFIX=FOC
 SEGNAME=ACCOUNTS, SEGTYPE=S1
  FIELDNAME=GL_ACCOUNT, ALIAS=GLACCT, FORMAT=A7,
   TITLE='Ledger,Account', FIELDTYPE=I, $
  FIELDNAME=GL_ACCOUNT_PARENT, ALIAS=GLPAR, FORMAT=A7,
   TITLE=Parent,
   PROPERTY=PARENT_OF, REFERENCE=GL_ACCOUNT, $
  FIELDNAME=GL_ACCOUNT_TYPE, ALIAS=GLTYPE, FORMAT=A1,
   TITLE=Type,$
  FIELDNAME=GL_ROLLUP_OP, ALIAS=GLROLL, FORMAT=A1,
   TITLE=Op, $
  FIELDNAME=GL_ACCOUNT_LEVEL, ALIAS=GLLEVEL, FORMAT=I3,
   TITLE=Lev, $
  FIELDNAME=GL_ACCOUNT_CAPTION, ALIAS=GLCAP, FORMAT=A30,
   TITLE=Caption,
   PROPERTY=CAPTION, REFERENCE=GL_ACCOUNT, $
  FIELDNAME=SYS_ACCOUNT, ALIAS=ALINE, FORMAT=A6,
   TITLE='System,Account,Line', MISSING=ON, $
```

CENTGL Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTGL   ON 05/15/03 AT 15.18.48

          ACCOUNTS
  01      S1
 **************
 *GL_ACCOUNT  **I
 *GL_ACCOUNT_>**
 *GL_ACCOUNT_>**
 *GL_ROLLUP_OP**
 *            **
 ***************
  **************
```

CENTSYSF Master File

```
FILE=CENTSYSF ,SUFFIX=FOC
 SEGNAME=RAWDATA ,SEGTYPE=S2
  FIELDNAME = SYS_ACCOUNT ,  ,A6    , FIELDTYPE=I,
   TITLE='System,Account,Line', $
  FIELDNAME = PERIOD      ,  ,YYM   , FIELDTYPE=I,$
  FIELDNAME = NAT_AMOUNT  ,  ,D10.0 , TITLE='Month,Actual', $
  FIELDNAME = NAT_BUDGET  ,  ,D10.0 , TITLE='Month,Budget', $
  FIELDNAME = NAT_YTDAMT  ,  ,D12.0 , TITLE='YTD,Actual',   $
  FIELDNAME = NAT_YTDBUD  ,  ,D12.0 , TITLE='YTD,Budget',   $
```

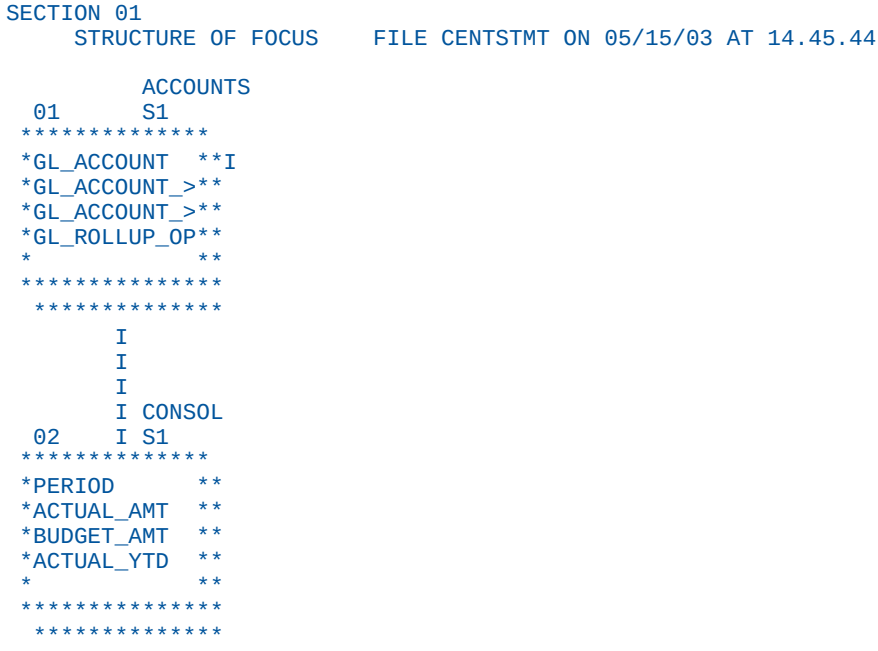
CENTSYSF Structure Diagram

```
SECTION 01
       STRUCTURE OF FOCUS    FILE CENTSYSF   ON 05/15/03 AT 15.19.27

          RAWDATA
  01      S2
 **************
 *SYS_ACCOUNT **I
 *PERIOD      **I
 *NAT_AMOUNT  **
 *NAT_BUDGET  **
 *            **
 ***************
  **************
```

CENTSTMT Master File

```
FILE=CENTSTMT, SUFFIX=FOC
 SEGNAME=ACCOUNTS, SEGTYPE=S1
  FIELD=GL_ACCOUNT, ALIAS=GLACCT,  FORMAT=A7,
   TITLE='Ledger,Account', FIELDTYPE=I, $
  FIELD=GL_ACCOUNT_PARENT, ALIAS=GLPAR, FORMAT=A7,
   TITLE=Parent,
   PROPERTY=PARENT_OF, REFERENCE=GL_ACCOUNT, $
  FIELD=GL_ACCOUNT_TYPE, ALIAS=GLTYPE, FORMAT=A1,
   TITLE=Type,$
  FIELD=GL_ROLLUP_OP, ALIAS=GLROLL, FORMAT=A1,
   TITLE=Op, $
  FIELD=GL_ACCOUNT_LEVEL, ALIAS=GLLEVEL, FORMAT=I3,
   TITLE=Lev, $
  FIELD=GL_ACCOUNT_CAPTION, ALIAS=GLCAP, FORMAT=A30,
   TITLE=Caption,
   PROPERTY=CAPTION, REFERENCE=GL_ACCOUNT, $
 SEGNAME=CONSOL, SEGTYPE=S1, PARENT=ACCOUNTS, $
  FIELD=PERIOD, ALIAS=MONTH, FORMAT=YYM, $
  FIELD=ACTUAL_AMT, ALIAS=AA, FORMAT=D10.0, MISSING=ON,
   TITLE='Actual', $
  FIELD=BUDGET_AMT, ALIAS=BA, FORMAT=D10.0, MISSING=ON,
   TITLE='Budget', $
  FIELD=ACTUAL_YTD, ALIAS=AYTD, FORMAT=D12.0, MISSING=ON,
   TITLE='YTD,Actual', $
  FIELD=BUDGET_YTD, ALIAS=BYTD, FORMAT=D12.0, MISSING=ON,
   TITLE='YTD,Budget', $
```

CENTSTMT Structure Diagram



Chapter 7

Error Messages

To see the text or explanation for any error message, you can display it online in your FOCUS session or find it in a standard FOCUS ERRORS file. All of the FOCUS error messages are stored in eight system ERRORS files.

- For z/OS, the ddname is ERRORS.

In this chapter:

Accessing Error Files

For z/OS, the error files are the following members in the ERRORS PDS:

- FOT004
- FOG004
- FOM004
- FOS004
- FOA004
- FSQLXLT
- FOCSTY
- FOB004

Displaying Messages

To display the text and explanation for any message, issue the following query command at the FOCUS command level

```
? n
```

where:

n

Is the message number.

The message number and text appear, along with a detailed explanation of the message (if one exists). For example, issuing the following command

```
? 210
```

displays the following:

```
(FOC210)    THE DATA VALUE HAS A FORMAT ERROR:
An alphabetic character has been found where all numerical digits are
required.
```

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

TIBCO, the TIBCO logo, the TIBCO O logo, FOCUS, iWay, Omni-Gen, Omni-HealthData, and WebFOCUS are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z