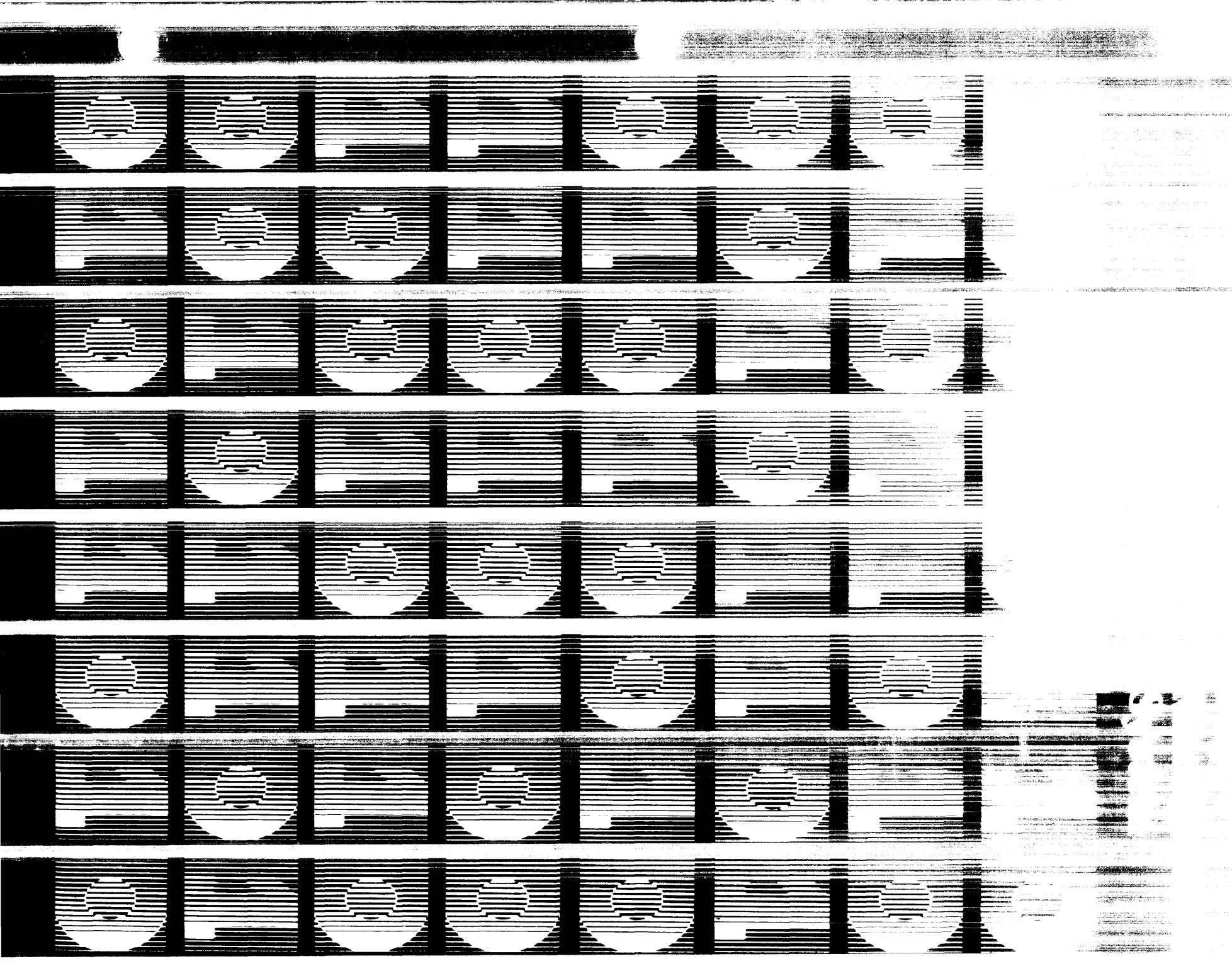


Printed in U. S. A.

DataGeneral
CS Commercial Systems

The GENAP™ System

File Management System

055-058-00

Data General Corporation, Westboro, Massachusetts 01580

TABLE OF CONTENTS

Chapter One

Introduction

The File Management System (FMS) is a database design tool that speeds and simplifies the development of COBOL applications systems. Through a series of interactive, menu-driven routines, the FMS helps you to design and modify hierarchical information structures. Then, the FMS writes Interactive COBOL source code for inclusion in your applications programs -- both to define the structures (Environment and Data Divisions) and to process the structured data (Procedure Division).

FMS Information Structures: Databases and Data Records

The File Management System's hierarchical scheme for organizing information is a "superstructure" built on Interactive COBOL's own data storage structures.

COBOL Data Structures

In a typical COBOL application, each datafile is structured as a collection of ~data records~. The data record (or "record", for short) is the unit of information for COBOL input/output statements: READ, WRITE, DELETE, etc. Each data record is a collection of ~elementary data items~, accessed individually or in groups.(Figure 1.01)

```
01 CUSTOMER-RECORD.
       03 CUSTOMER-KEY.
              05 ID-NUMBER  PIC X(5).
       03 CUSTOMER-DATA.
              05 NAME       PIC X(20).
              05 ADDRESS    PIC X(25).
```

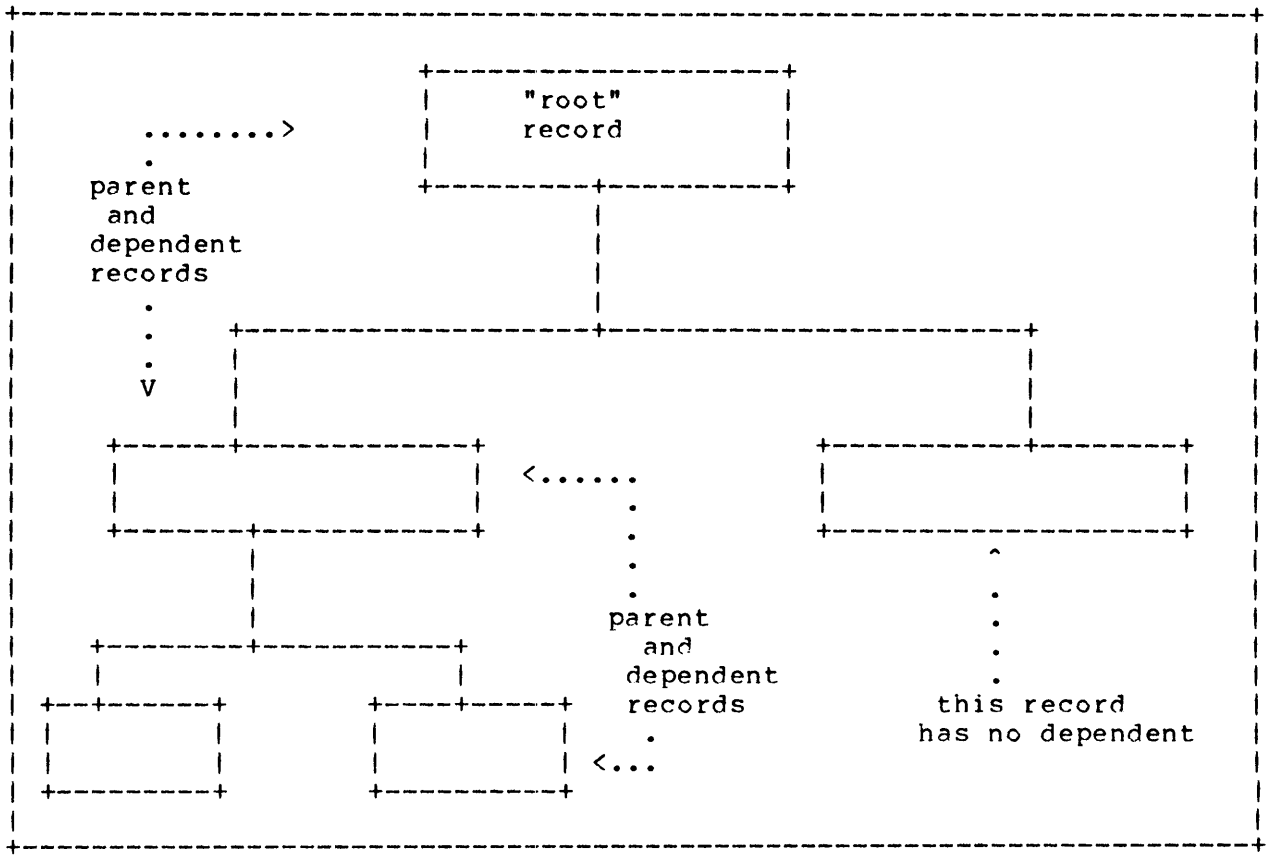
(1.01) A SIMPLE INTERACTIVE COBOL RECORD STRUCTURE. Interactive COBOL record structures include elementary data-items organized into groups.

Many COBOL applications use ~indexed datafiles~. Individual data records are created in and retrieved from these files using the value of a RECORD KEY field.

FMS Data Structures

The FMS starts where COBOL's indexed file structures leave off. COBOL allows the grouping of elementary data-items into structured records -- the FMS allows the grouping of records into hierarchical structures called "databases". Each database structure is organized as a "tree".(Figure 1.02) Some record types in the tree structure ("parents") logically encompass one or more other record types ("dependents"). One record type acts as the "root" of the tree.

Each record in a database structure is distinguished from other records of the same type by a "local key" value. Using a key-concatenation scheme, the FMS creates a larger key that uniquely identifies the record throughout the entire database. This structured key also establishes the hierarchical tree structure relating the database records.



(1.02) THE STRUCTURE OF AN FMS DATABASE. The File Management System organizes COBOL records into "trees": programmer-defined hierarchical structures.

The FMS makes it easy for you to use more than one of these hierarchical databases in any of your applications programs; you need not try to organize all of the data to be processed by your application into one giant tree. It is also easy to use the same FMS database in several different programs.

Programming with the FMS: An Overview

Since the FMS is a source-code generator, implementing a FMS-based program comprises two major steps:

1. Using the FMS to generate the source code that defines and accesses one or more databases.

2. Writing programs that incorporate the FMS-written code.

To give you a better idea of FMS usage, we present a quick look at what these steps entail. (Chapter 3 presents a comprehensive explanation of the various FMS routines, including complete operating instructions.)

Step 1: Generating FMS Source Code

With the File Management System's "maintenance" routines, you enter database specifications using interactive data-entry screens. Instead of generating source code directly, the FMS stores these specifications in work files. This allows you to change the specifications quickly and easily.(Figure 1.03) This is particularly valuable if (most likely, ~when~) it becomes necessary to fine-tune your information structures, or to significantly revise them.

```
                        DATA ELEMENT MAINTENANCE

  1. COBOL NAME                    ID-NUMBER

  2. PICTURE                       X(20)

  3. USAGE (D)ISPLAY/(C)OMP

  4. INITIAL VALUE

  5. NOTES

                                   (1.) ONLY PRIMARY KEY FIELD
```

(1.03) USING FMS MAINTENANCE ROUTINES. With FMS maintenance routines, you interactively create and modify the specifications that define FMS database structures.

The specifications you enter and modify using the maintenance routines define the following:

* DATA ELEMENTS. These are, essentially, COBOL elementary data-items. The FMS maintains a "data element dictionary" of all the data-items you define.

* RECORD TYPES. These are COBOL data record descriptions. Each record may include one primary key, one alternate key, and multiple non-key data-items.

* DATABASE STRUCTURES. Each database structure is a group of record types organized into a tree structure by your specifications of parent record type/dependent record type relationships.

After you have completely defined a database structure using the maintenance routines, you use the File Management System's code-generation routines to generate source code COPY files that define the structure in the Environment and Data divisions. (Figure 1.04)

In addition to generating these database definitions, the FMS generates a set of subroutines (Procedure Division paragraphs) with which your program processes the database. The FMS procedures perform all standard input/output functions, including:

* READ a record (by primary key or alternate key, or sequentially).

* WRITE a new record (by primary key).

* REWRITE a modified record (by primary key).

* DELETE (using Interactive COBOL's "logical deletion" facility) an existing record.

* LOCK and UNLOCK records.

* (optional) Provide an interface to a programmer-supplied record "logging" procedure.<*>

<*> With many applications, it is necessary to keep a record of all modifications to the database for purposes of auditing transactions, and for restart and recovery. This record-keeping activity is called ~logging~. The FMS optionally produces code that produces before-and-after images of all transactions that alter a database. The actual logging procedure that processes these images must be programmer-supplied.

Appendix B lists the various source code modules that the FMS writes to define and access a database.

Step 2: Writing Programs that Incorporate FMS Code

For the most part, you won't have to change your programming style to write a program that uses FMS databases. It's still up to you to implement data-entry screen formats, build report lines, perform calculations, etc.(Figure 1.04) The File Management System's code handles only the disk input/output tasks, moving information between disk storage and main memory on a record-by-record basis.

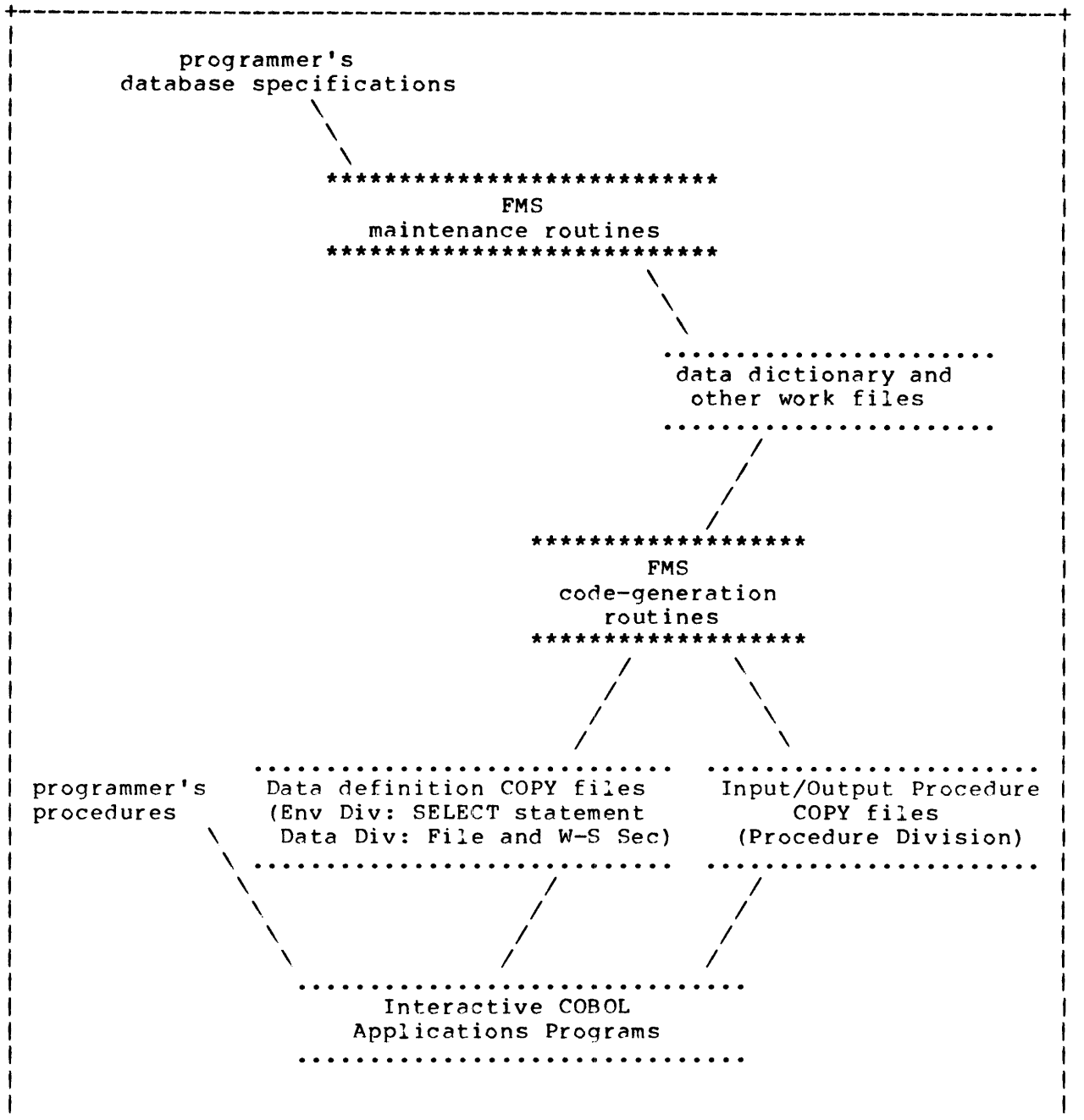
After the FMS has produced source code to define and process a database, how does the code find its way into your program(s)? Since the FMS places most of the code it creates in separate files, it is easy to use the COPY statement to incorporate the code into your programs at compilation time. In fact, the FMS provides the further service of writing sets of COPY statements which you can insert directly into your programs, using either CRT/EDIT or IC/EDIT commands.

Chapter 4 includes complete instructions for incorporating FMS-written code into your applications programs.

An Example of FMS-Based Processing

To illustrate how the major parts of FMS-written code fit together, here is how an operator might modify a data record using a FMS-based data-entry program. We present the flow of program logic as a chart to emphasize the division of labor between the FMS and the programmer:

Programmer Defined Procedures	FMS-written I/O Routines
1. A display screen format ACCEPTs values for the key fields that identify the record. These values are stored in Data Division areas defined by FMS-written code.	
2. A PERFORM statement calls an indexed-READ routine.	2A. READ routine
3. A display screen format DISPLAYs the values of the fields retrieved from disk storage and ACCEPTs modifications by the program operator.	
4. A PERFORM statement calls an indexed-REWRITE routine.	4A. REWRITE routine.
5. (optional) A PERFORM statement calls a programmer-written logging routine.	5A. (optional) Before-and-after images of the modified record are moved to FMS-named (but programmer-defined) storage area for use by a logging routine.



(1.04) USING FMS CODE-GENERATION ROUTINES. The FMS stores your database specifications in workfiles until you order it to write source code to the specifications. Then, you incorporate this code into your applications programs.

Chapter Two

FMS Data Structures

This chapter describes FMS data structures, including both the "virtual" structures seen by the programmer and the actual structures implemented with Interactive COBOL indexed files. Also included is a description of the flow of data in an FMS-based program, both to/from disk storage and between the File Section and Working-Storage areas of main memory.

Structures vs. Data

In this chapter, we will be careful to distinguish between the ~structures~ that define the format of data storage and the actual ~data~ stored in these structures. For example, a standard Interactive COBOL indexed data file includes one type of record, but may include hundreds or thousands of actual data records. Each of these records has the same RECORD KEY field (for instance: EMPLOYEE-NUMBER), but each has a different RECORD KEY value.

We will use the terms ~database structure~ and ~record type~ to indicate the structures into which data is formatted. The terms ~database~ and ~record~ will refer to the actual collections of data which "fill in" these formats.

The Database Example

In order to illustrate FMS database structures, we will pursue throughout this chapter a simplified example of a hierarchical database.

Suppose a manufacturer wants to develop a simple bookkeeping system to keep track of all his customers, along with all their orders and payments. This information naturally falls into three record types:

* Each CUSTOMER RECORD stores overall information concerning a single customer.

* Each PURCHASE RECORD stores a single purchase transaction. For simplicity, let's restrict each purchase transaction to one item of the manufacturer's stock.

* Each PAYMENT RECORD stores a single payment transaction. For simplicity, again, let's assume that each payment transaction corresponds to a particular invoice sent by the manufacturer.

Conceptually, each purchase record and payment record pertains to a particular customer. So it's sensible to make the CUSTOMER RECORD type the "root" of the database structure. In FMS parlance, we say that each purchase record and each payment record is "dependent" on a particular customer record. We can also say that a customer record is the "parent" of a group of purchase records and a group of payment records.

Armed with this example, let's proceed to an examination of the way in which the FMS organizes information.

Virtual Data Structures

In this section, we present the structure of an FMS database as it appears when you are using the FMS itself. We will ignore (for the time being) some of the implementation details that go into actual FMS-written code. These details come into play later, when you use FMS code in your programs.

An FMS ~database~ structure is a hierarchy of ~data record~ types. Each record type encompasses groups of ~data elements~.

Interactive COBOL supports the storing of a virtually unlimited number of actual data records of each type. In practice, the amount of data that a database can store is limited by the capacity of the system's on-line disks.

Data Elements

A ~data element~ is an Interactive COBOL data-item. The principal job of a data element is to assign a data-name to a data storage area, and to describe the length and type of the storage area with a PICTURE clause. Since Interactive COBOL includes both group data structures and table structures, some data elements act as group headers, while others include the REDEFINES and OCCURS clauses to define tables.

In its "Define/Maintain Lata Elements" routine, the FMS prompts you to specify the following parts of a data element (Figure 2.01):

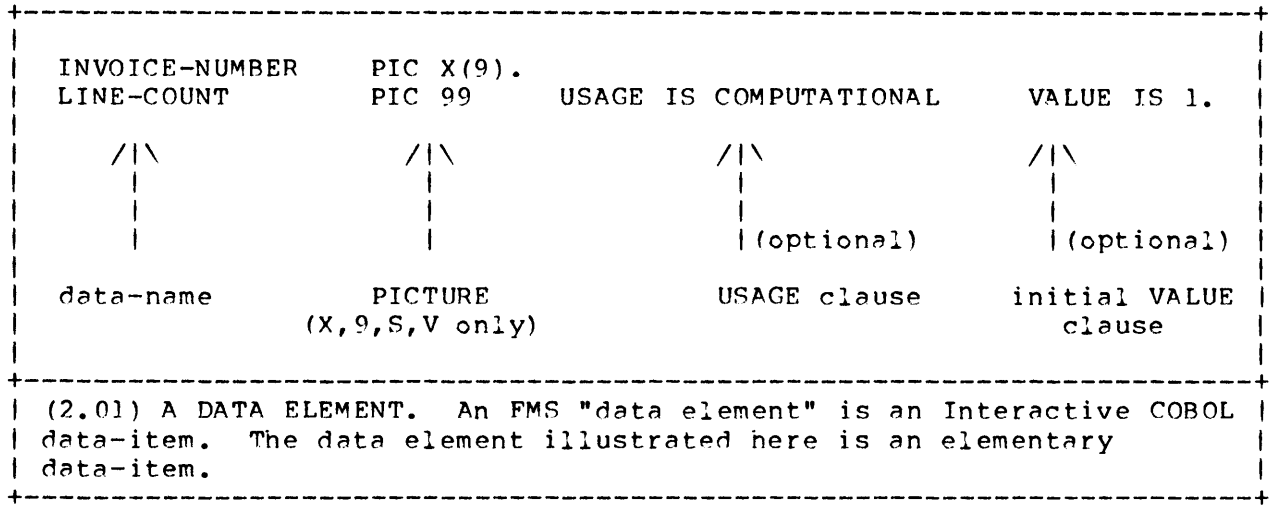
COBOL name: A standard COBOL data-name.

PICTURE: A standard COBOL PICTURE clause, specifying the type and number of characters in the data element.

USAGE: (optional, and only for data elements with numeric PICTUREs) A "USAGE IS COMPUTATIONAL" clause.

VALUE: (optional) A value to be stored in the data element at the start of program execution.

The FMS stores these specifications in the "Data Element Dictionary" for use later, when you construct data records from the elements.



Data Record Types

An FMS data record type is functionally identical to an ordinary COBOL data record type: it is a structured collection of elementary data-items. As in other COBOL situations, the record is the unit of information processed by FMS input/output routines: A "RETRIEVE" routines reads a record from disk storage into main memory; a "REPLACE" routine rewrites an existing record to disk; and so on.

In its "Define/Maintain Record Elements" routine, the FMS prompts you to build data record types element-by-element.(Figure 2.02A) When you assign a data element to a record type, you supplement the basic data description with specifications for how the element will be used:

* PRIMARY KEY - A single data element, or a group of elements, may act as the RECORD KEY for the record structure.

* ALTERNATE KEY - A single data element, or a group of elements, may act as the ALTERNATE RECORD KEY for the record structure. (The FMS allows only one alternate key for a particular record type.)

* NON-KEY ELEMENT - Additional data elements, either singly or in groups, define additional storage areas in the record structure.

* GROUP HEADER - A data element may act as the name of a group. The FMS allows the grouping of primary key fields, alternate key fields, and non-key fields. In the "Define/Maintain Record Elements" routine, you enter COBOL level numbers to define groups.

* "REDEFINES" or "OCCURS" ITEM - the FMS can include one of these clauses in a data item, allowing you to define tables. For example:

```
        03 MONTHS-STRING PIC X(24) VALUE IS "JAFEMRAPMYJNJYAUSEOCNODE".
        03 MONTHS-TABLE REDEFINES MONTHS-STRING.
                05 MONTH-CODE PIC X(2) OCCURS 12 TIMES.
```

Using this assignment of data elements, the FMS codes a data description for the record type in the form illustrated in Figure 2.02B. Note in this illustration that the FMS includes additional fields that establish the position of the record type in a database hierarchy. You need not be concerned with these extra fields when you are using the FMS to define record types. We'll discuss them in the "Actual Data Structures" section of this chapter and in Chapter 4, "Programming with FMS Code".

A. FMS Record Definition Routine

ITEM	LEVEL	NAME	KEY	OCCURS	REDEFINES	LENGTH
01	03	CUSTOMER-KEY	Y			
02	10	ID-NUMBER	Y			
03	03	CUSTOMER-DATA	A			
04	10	NAME				

B. Typical FMS record structure

```
01 PAYMENT-RECORD.
   03 ID-NUMBER        PIC X(5).                    <---- level-0 key
   03 FILLER           PIC 9(02) COMP VALUE 03.
   03 INVOICE-NUMBER   PIC X(9).                    <---- level-1 key
   03 PO-NUMBER        PIC X(7).                    <---- alternate key
   03 FILLER           PIC X(13) VALUE LOW-VALUES.
   03 AMOUNT           PIC 99999V99.                <---- non-key
                                                          element
```

(2.02) DEFINING A RECORD TYPE. (A) You interactively create and modify record types by grouping previously-defined data elements. (B) An FMS record type includes elements that define a primary local key, an (optional) alternate key, and non-key data fields.

Databases

A database structure is a tree-like hierarchy that includes 1-48 record types. Figure 2.03 illustrates the following points.

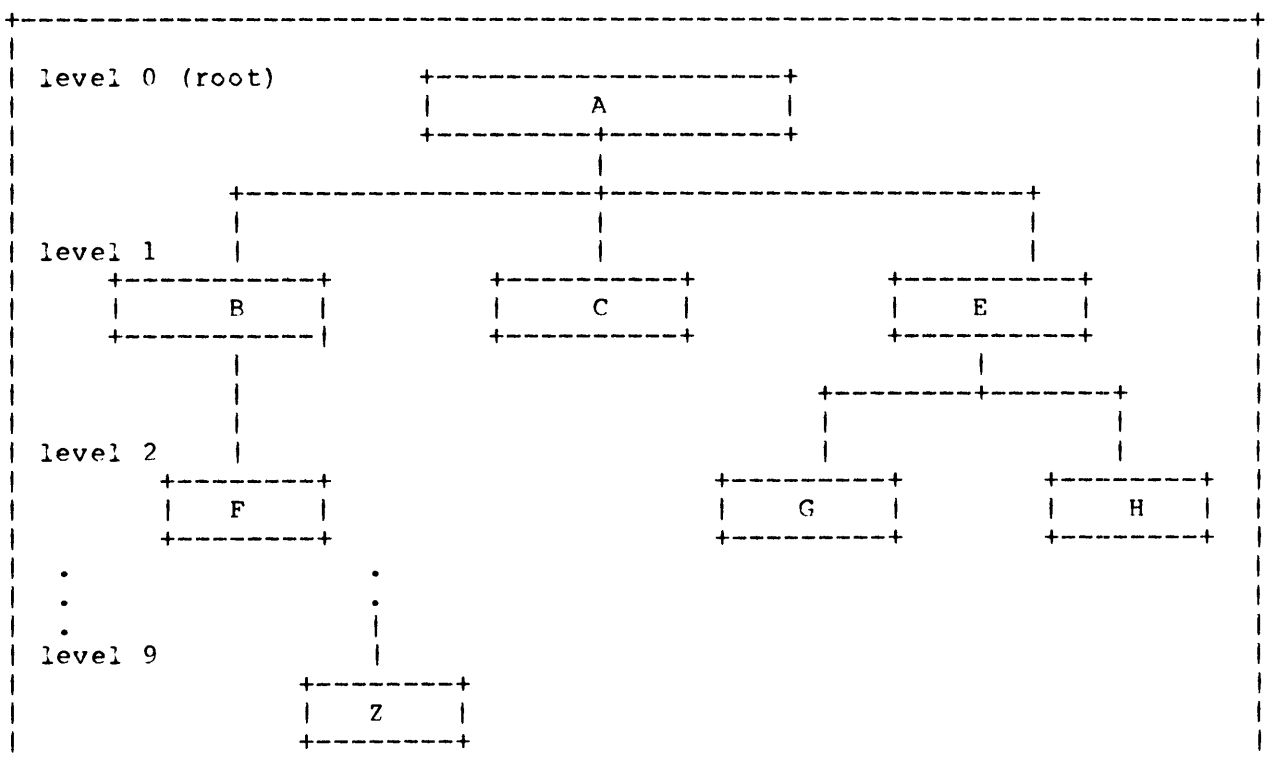
* Record types in a database tree structure are related in pairs as "parents" and "dependents".

* One record type, the "root" of the database structure, is unique. The root is the only record type that has no parent.

* All other record types in the hierarchy have exactly one parent, so that there is an unambiguous path from any record type back to the root.

* You may assign some record types many dependents, while others may have none -- you need not create a "balanced" tree.

* The root record type is said to be at "level 0" of the hierarchy. Record types that have the root as their parent are said to be at "level 1" of the hierarchy; record types that have a level-1 record as their parent are said to be at "level 2" of the hierarchy, and so on. The tree may proliferate up to "level 9".



(2.03) THE HIERARCHICAL DATABASE STRUCTURE. An FMS database structure comprises record types organized into a multi-level tree.

Much of what we have said about the database ~structure~ applies equally to the actual records that make up a working database. For instance, we may say that "each record has exactly one parent" and, in our example, that "a PAYMENT-RECORD is at level 1 of the database". Note, however, that whereas there is a single record TYPE at the root of the database structure, there will be many different actual root records. The database ~structure~ is a single tree, but the actual database is a collection of trees, each branching from a different root record.

Keys to Record Types

Included among the data elements that make up a record type are one or more elements that make up the "key" to the record type. An individual data record is identified by its key -- that is, it is distinguished from all other records of the same type with the same parent.

A database structure includes many record types, each type with its own key. We will often use the term "local key" to describe a particular record type's key.

Using this terminology, we can summarize as follows:

> If two records of the same type have the same parent, the values of their local keys must be different.

This rule implies the valuable flexibility that the FMS affords in building databases (Figure 2.04):

* If two records with the same parent are of different types, they may both have the same key value.

* If two records of the same type have different parent records, they may both have the same key value.

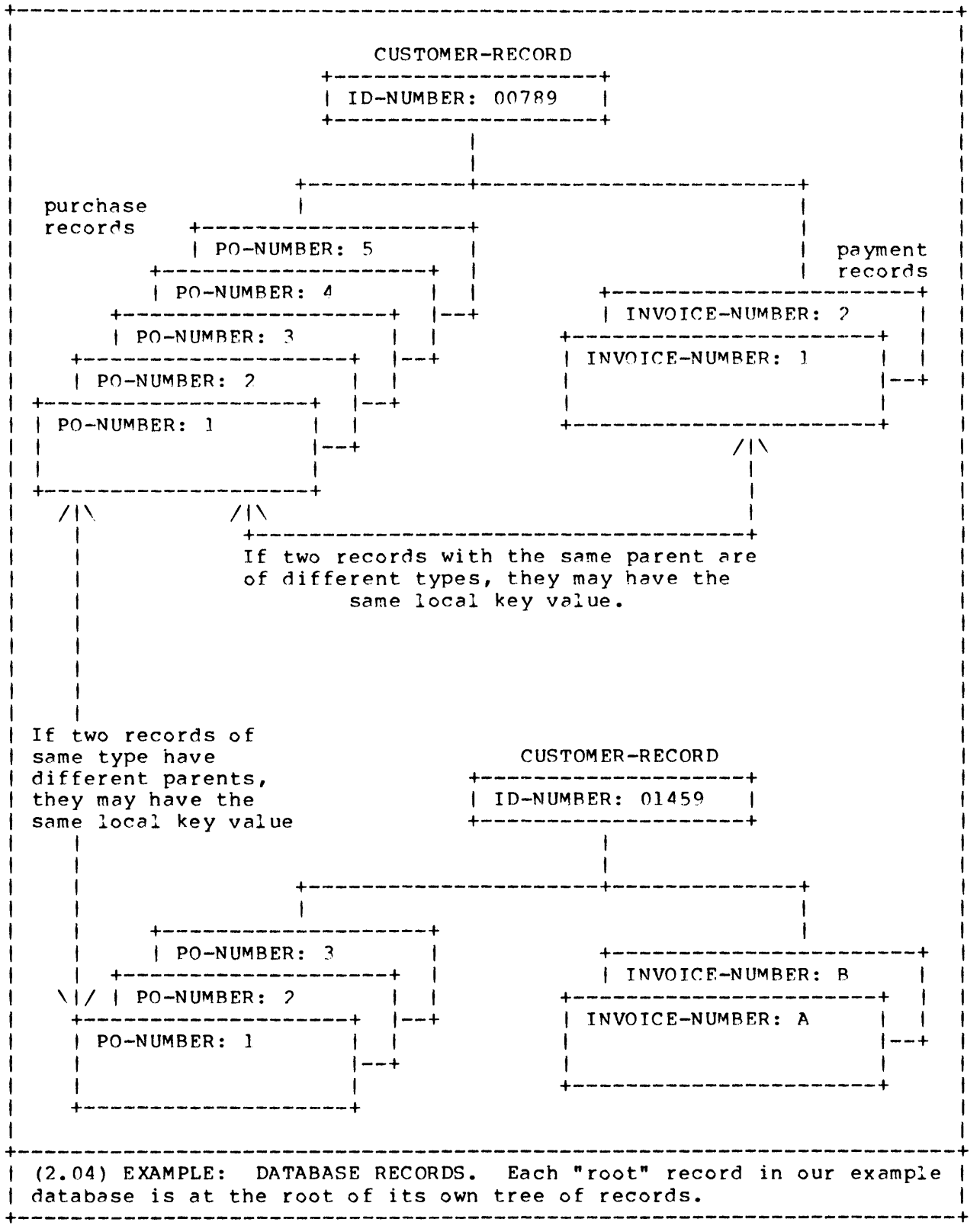
Alternate Keys

The FMS uses Interactive COBOL's ~alternate key~ feature as well as its ~primary key~ feature. You may specify data elements in each record type to make up the "alternate key" to records of that type.

Keyless Records

For completeness, we mention here a special record type that can help conserve disk storage space. Any record type in the database structure may have one or more "keyless record" types as dependents. Unlike other record types, there may be only one actual record of each keyless type. Since there is only one instance of this record type, no local key element is required to identify it -- hence, the term "keyless." Functionally, a keyless record acts as an extension, an extra field in the parent record.

For details on how keyless records can save disk space, see the discussion in the "Actual Data Structures" section that follows.



(2.04) EXAMPLE: DATABASE RECORDS. Each "root" record in our example database is at the root of its own tree of records.

Actual Data Structures

In this section, we examine the way in which the FMS implements the hierarchical database structure using Interactive COBOL indexed data files. The essence is a key-concatenation scheme involving the local keys of the record types.

COBOL Indexed Files and Records

Each FMS database is physically realized as a set of one or more indexed data files. A typical Interactive COBOL indexed file contains many records structured according to a single File Section data description. One field (single or group) in the data description is the RECORD KEY. Each record has a unique RECORD KEY value which identifies the record -- no two records may have the same value stored in their RECORD KEY fields.

FMS Record Structure -- Single File Implementation

In a simple FMS database, all records are stored in a single indexed file. (Multiple-file implementation is discussed below in "Space-Saving Implementation Features".) The FMS creates a record description for this indexed file that includes:

1. RECORD KEY area with enough space to accommodate a sequence of local key values.

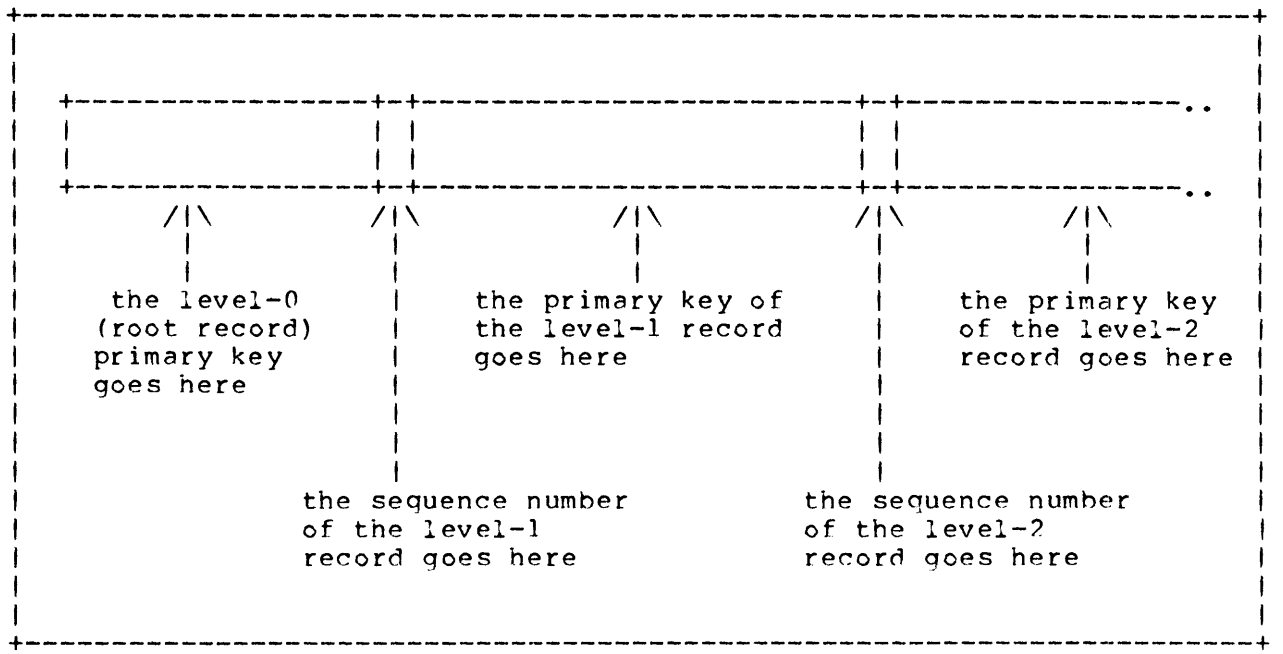
2. ALTERNATE KEY area for loading a single alternate-key value.

3. FILLER area for holding each record's non-key fields.

The Concatenated RECORD KEY

Using your database specifications, the FMS manufactures a RECORD KEY that includes "slots" for all the local keys needed to locate a record within the hierarchy.(Figure 2.05) By "concatenating" (linking together) the local key values in a direct line from the root record down to a particular descendant record the FMS creates a ~concatenated key~ that uniquely identifies that particular record in that database.

THE SEQUENCE NUMBER PREFIX. To ensure uniqueness of the concatenated RECORD KEY for each record, the FMS adds a bit of overhead.(Figure 2.05) The FMS identifies each record type in a database by a "sequence number." It uses this sequence number as a prefix to each local key value when it constructs a RECORD KEY value. This prevents accidental key duplication.



(2.05) THE CONCATENATED RECORD KEY. By concatenating local record keys the FMS builds a RECORD KEY for the indexed file that stores the entire database.

Our example will illustrate this point. Suppose that both a PURCHASE-RECORD and a PAYMENT-RECORD for John Smith have local key values of "00101". Since the two dependent records have different sequence numbers, FMS-written routines can construct different RECORD KEY values:

level-0 key (ID-NUMBER)	seq. # prefix	level-1 key	
John Smith	2	00010	<------ (INVOICE-NUMBER)
John Smith	3	00010	<------ (PO-NUMBER)

The Alternate Key Area

If you specify an alternate key for any of the record types in a database structure, the FMS reserves an area in the data record just after the RECORD KEY area. Whenever your program performs an alternate key access of the data file, FMS-written code moves the value of the alternate key to this area before performing the input/output routine. (For alternate key access, the FMS does not include a duplication-avoidance mechanism comparable to the sequence-number-prefix scheme explained above. Thus, you'll have to perform your own validation routines to prevent unwanted duplication of alternate key values.)

The FILLER Area

The indexed record definition includes enough FILLER to accommodate the non-key portion of any single record in the database.

(Recall that in a single-file implementation, all records in the database are stored in a single indexed file. FILLER allocation for a multiple-file implementation is discussed in section "Multiple-File Databases" below.)

Actual Record Coding

Figure 2.06A shows the File Section record description for the purchases/payments database. The FMS creates this generic record description to act as a "loading/shipping dock" for disk input/output of all records in the database. Where are the data-names? ... in Working-Storage. The FMS writes routines that automatically move records between this File Section record area and work areas it defines in Working-Storage -- a different area for each record type. In these work areas, FMS code defines and processes individual data elements.<*>

For your programming purposes, you may treat the work areas as if they were actual record descriptions. You never need to address the File Section record area directly.

The Working-Storage definition of PURCHASE-RECORD in our example (Figure 2.06B) includes counterparts to the three File Section areas described above. A comparison with Figure 2.06C shows that the FMS has added to the programmer's definition of PURCHASE-RECORD "extra fields" to accommodate its concatenated (primary) key and its alternate key:

* The RECORD KEY area includes space for the local key of its parent, CUSTOMER-RECORD. It also includes FILLER padding of the local key area for level 1; the PURCHASE-RECORD's key is seven bytes long, but the PAYMENT-RECORD's key is nine bytes long.

* Though the PURCHASE-RECORD has no alternate key, the FMS has reserved an alternate key area of 20 bytes. The FMS codes the same size alternate key area in every record type in the database structure.

* The non-key area, coded as FILLER in the File Section generic record, reflects the data-element structure: STOCK-ITEM, QUANTITY, etc.

<*> This subject is treated in greater detail below: "FMS Usage of the File and Working-Storage Sections".

(A) File Section: actual record description

```
01 PURPAY-RECORD.                              . }
  02 PURPAY-KEY.                               . }
    03 LEVEL-1.                                . }
      04 LEVEL-0.                              . }  RECORD KEY area
        05 KEY-LEVEL-0  PIC X(05).              . }
      04 PREFIX-LEVEL-1 PIC 9(02) COMP.        . }
    03 KEY-LEVEL-1      PIC X(09).              . }
                                                .
  02 PURPAY-ALTERNATE-KEY-1   PIC X(20).        .    alternate key area
                                                .
  02 FILLER                   PIC X(040).       .    non-key area
```

B) Working-Storage Section: work area

```
01 PURCHASE-RECORD.                                .
   03 ID-NUMBER  PIC X(5).                          <-- level-0 local key
   03 FILLER    PIC 9(02) COMP VALUE 02.            <-- seq. number prefix
   03 PO-NUMBER  PIC X(7).                          <-- level-1 local key
   03 FILLER     PIC X(02) VALUE LOW-VALUES.

   03 FILLER PIC X(20)     VALUE LOW-VALUES.  <-- alternate key area

   03 STOCK-ITEM    PIC X(10).
   03 QUANTITY      PIC 999.                        non-key elements
   03 UNIT-PRICE    PIC 9999V99.
   03 TOTAL-PRICE   PIC 99999V99.
```

(C) Record as defined by FMS programmer

```
01 PURCHASE-RECORD.
   03 PO-NUMBER    PIC X(7).          <--- level-1 local key
   03 STOCK-ITEM   PIC X(10).         }
   03 QUANTITY     PIC 999.           }  non-key elements
   03 UNIT-PRICE   PIC 9999V99.       }
   03 TOTAL-PRICE  PIC 99999V99.      }
```

(2.06) EXAMPLE: RECORD DEFINITION CODING. A virtual record defined by the programmer is implemented with two storage structures: a "generic" record (File Section) and a "specific" work area (Working-Storage Section).

Record Sequence

To further illuminate the File Management System's key-concatenation scheme, let's take a look at how the records of a database are actually stored in a single indexed file. For this purpose, we augment the example database structure with two level-2 record types, LINE-ITEM and DELIVERY, both dependent on PURCHASE-RECORD.(Figure 2.07A)

Figure 2.07B shows both the conceptual relationships and the actual order of records pertaining to two customers. Note that:

* The entire tree of records dependent upon a particular root record is stored just after the root record.

* At each FMS level, all dependent records are stored just after their parent.

* When a record type has several dependent types, the dependent records are grouped by type. The order of the groups is determined by the sequence-number prefixes.<*>

Space-Saving Implementation Features

The basic FMS strategy uses one indexed file to store all records in a database. However, these records may have various, perhaps vastly different lengths. This can result in significant wastage of disk storage, as the Purchases/Payments database records shown in Figure 2.08 illustrate.

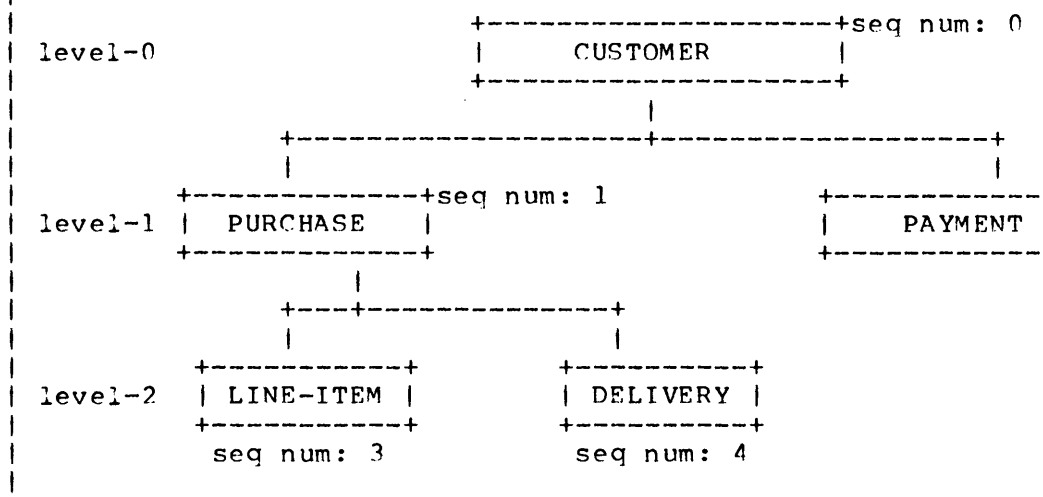
The FMS offers two solutions to this problem, ~keyless records~ and ~multiple-file databases~.

Keyless Records

If one record type is considerably longer than others in the same database structure, you may want to split off one or more non-key data elements as a special kind of dependent record type, a ~keyless record~. In our purchases/payments example, we can split off the COMMENTS element, as illustrated in Figure 2.09. A keyless record is a dependent record that acts as an extension of its parent. With keyed record types, many actual records may be created, distinguished from each other by local key values. But there may be only one record of a keyless type. Since it is unique, no local key is required in the record definition -- once the parent record has been identified, so has its keyless dependent.

<*> The FMS assigns sequence numbers to record types automatically. You may use the "Resequence Data Base Records" routine to reassign these numbers and, hence, change the order in which records will be stored on disk.

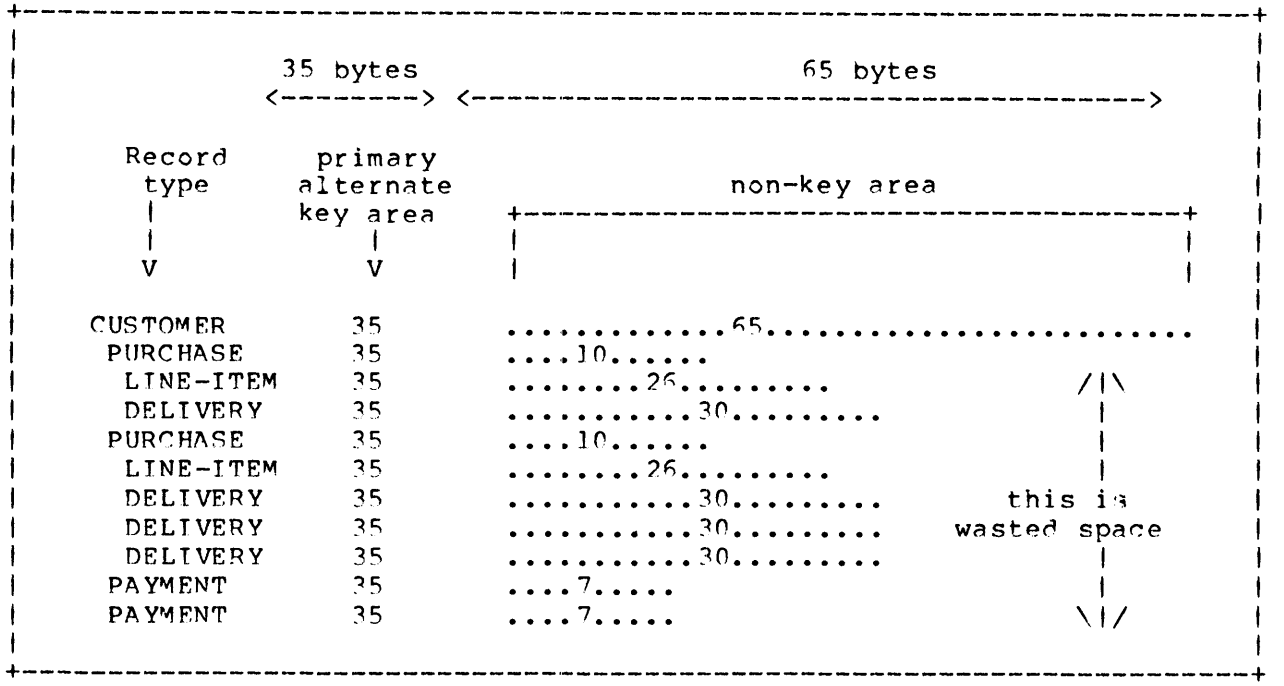
(A) Database Structure



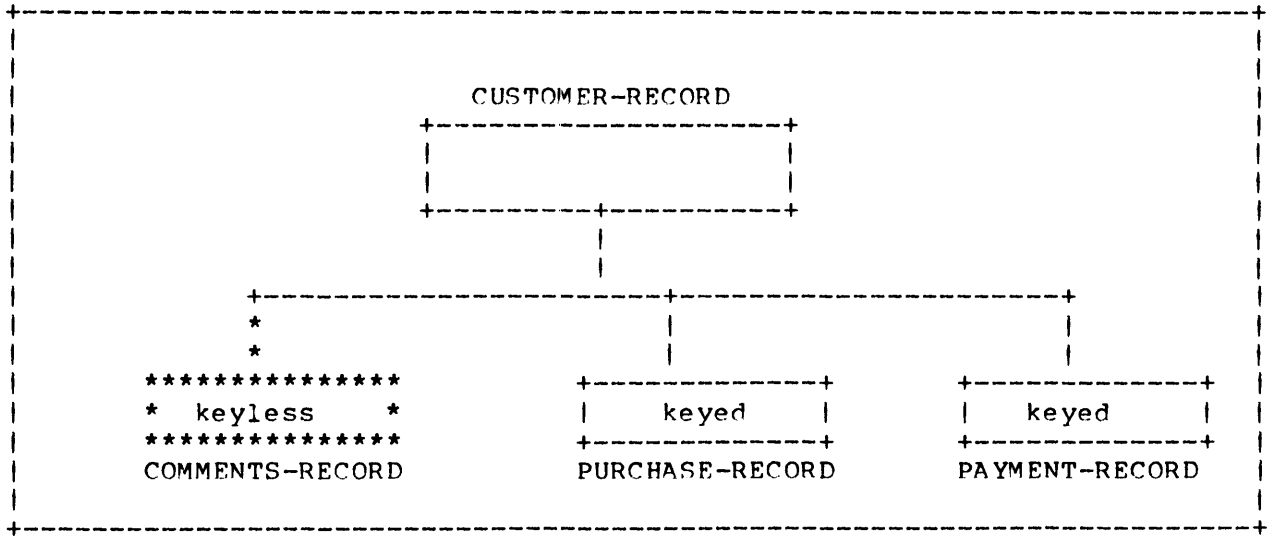
(B) Record Sequence

Record type	Level-0 key	Sequence number prefix	Level-1 key	Sequence number prefix	Level-2 key
CUSTOMER	00789	.	-------	.	-------
PURCHASE	00789	1	PU974	.	-------
LINE-ITEM	00789	.	PU974	3	CX-0873
DELIVERY	00789	.	PU974	4	DEL001
PURCHASE	00789	1	PU1887	.	-------
LINE-ITEM	00789	.	PU1887	3	CY-9913
DELIVERY	00789	.	PU1887	4	DEL001
DELIVERY	00789	.	PU1887	4	DEL002
PAYMENT	00789	2	XYZ*432	.	-------
PAYMENT	00789	2	XYZ*761	.	-------
CUSTOMER	01459	.	-------	.	-------
PURCHASE	01459	1	556AOZ	.	-------
LINE-ITEM	01459	.	556AOZ	3	RW-1
LINE-ITEM	01459	.	556AOZ	3	RW-3
DELIVERY	01459	.	556AOZ	4	RZ-23
PURCHASE	01459	1	775BOZ	.	-------
LINE-ITEM	01459	.	775BOZ	3	SS-04
LINE-ITEM	01459	.	775BOZ	3	OA-10
PAYMENT	01459	2	ABC*098	.	-------

(2.07) EXAMPLE: RECORD ORDER. Our (expanded) example illustrates how the concatenated RECORD KEY orders the records in a database.



(2.08) SPACE WASTAGE. If a database stores records of widely-varying sizes, some disk storage may be wasted.

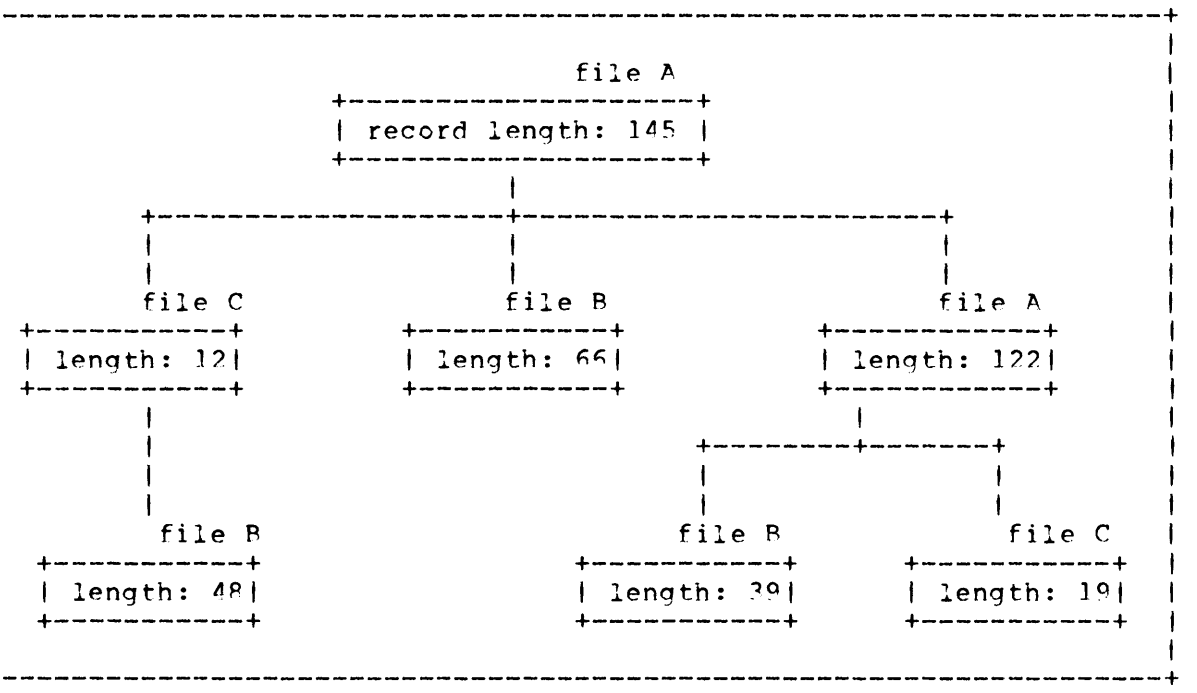


(2.09) KEYLESS RECORDS. Splitting off part of a record as a "keyless record" dependent can help to equalize the sizes of the database's record types, minimizing disk-space wastage.

Multiple-File Databases

THe FMS can implement a particular database structure as several indexed data files. During the "Define/Maintain Database Records" routine, you may assign record types to different data files. For each file, the FMS automatically creates a "generic" record description of optimal size for each group of record types. The record description includes enough FILLER to accommodate the non-key portion of any record assigned to that file -- it need not handle records assigned to other files.

If you assign record types to different data files carefully, you can significantly reduce the amount of disk space wasted as a result of widely-varying record sizes in the same database.(Figure 2.10)



(2.10) MULTIPLE-FILE DATABASES. Distributing the records in a database among several files can help to conserve disk storage space.

Sequential-File Databases

Though the FMS is fundamentally oriented toward indexed-file implementations, you may elect to use one or more sequential files to store the records in a database. This strategy saves disk space: the system does not need to maintain an index structure for a sequential file. But you lose the indexed-sequential (ISAM) access capability upon which many business applications rely -- sequential files may only be accessed sequentially.

If you choose to implement a database as a sequential file, the actual record order, explained above in "Record Sequence", becomes a critical factor in application performance.

FMS Usage of the File and Working-Storage Areas

In the preceding section, we saw that the FMS creates two storage structures for each record type you define with the "Define/Maintain Database Records" routine, one in the File Section and one in Working-Storage. In this section, we examine further the way in which procedures -- both the FMS's and your own -- use these storage areas. For simplicity, the discussion below assumes that the entire database is stored in a single indexed file.

We may refer to the two storage areas created for each record type as the "generic" and "work" areas:

* The "generic" area in the File Section is the actual record area for the indexed file storing the database. It is the database "loading dock": When a keyed READ, WRITE, DELETE, etc., statement is executed, the system uses the RECORD KEY defined in the generic area to identify the record to be accessed. The system READs records from disk into this area; and it WRITEs records from this area to disk.

Although the generic area is "where the action is," you never address this area directly. The names of the data-items are all FMS-created and bear no relation to those of the data elements you've defined.

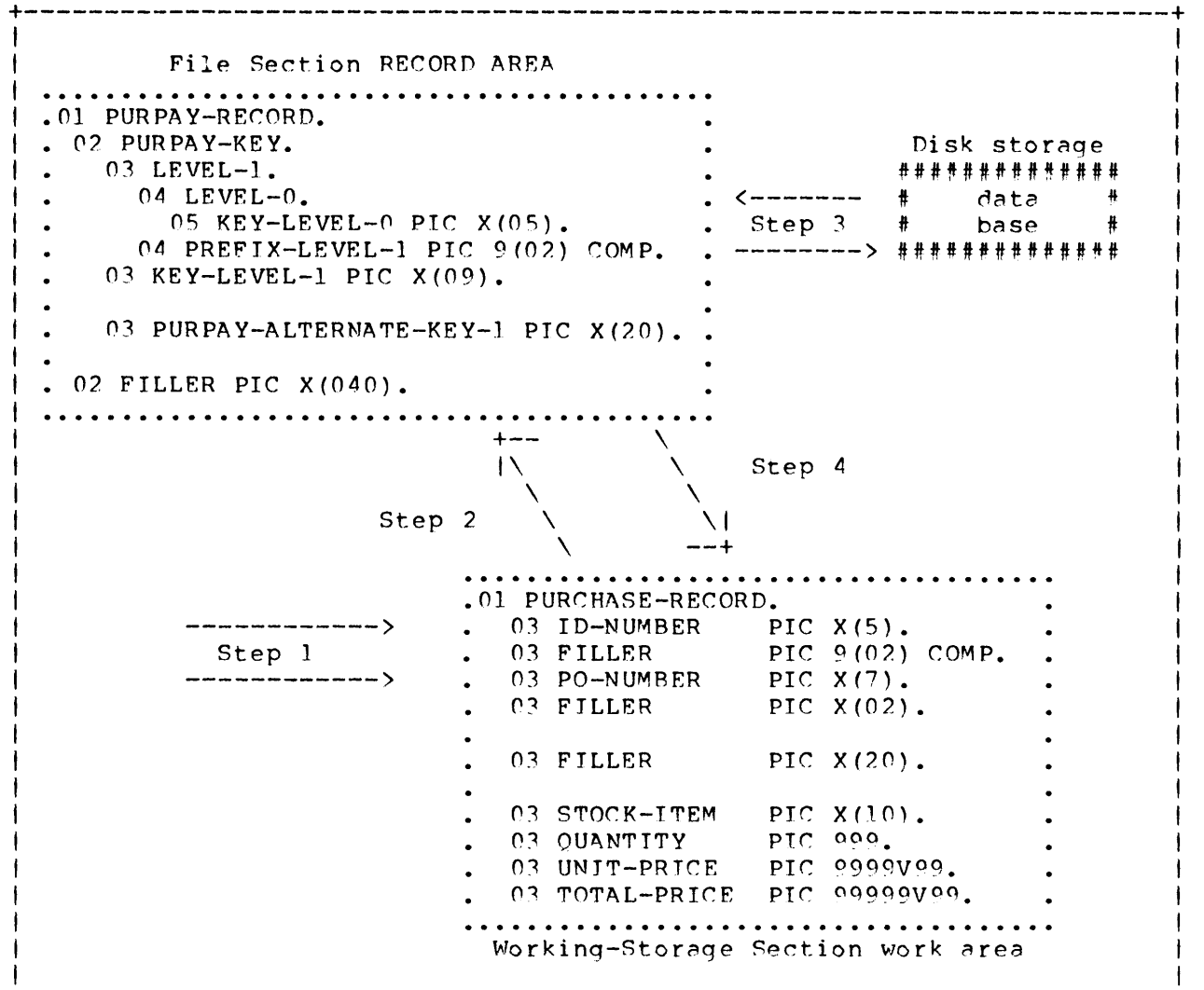
* A "work" area in the Working-Storage Section is created for each record type. The data elements you named in the "Define/Maintain Record Elements" routine are coded in these work areas. Accordingly, you manipulate data by addressing these work areas with your procedures.

For most purposes, you may code as if the work areas really were File Section RECORD AREAs for the record types. FMS-written procedures automatically move data between the generic area and the individual work areas. Figure 2.11 illustrates the flow of information when we READ a record from our purchases/payments database. More generally, an input/output task comprises the following steps.

1. You load values into the key elements defined in the work area. For a primary key access, this includes loading a set of local key values that make up the concatenated RECORD KEY that will actually be used to access the database.

2. You call an FMS-written procedure. The procedure moves the contents of the work area to the generic record area, executes an input/output statement, and (if appropriate) moves newly-retrieved data from the generic area back into the proper work area.

3. You test the value of an FMS-maintained data-item to determine whether the input/output attempt was successful, and branch accordingly.



Step 1: Load the work area key fields.

Step 2: Call the FMS-written procedure RETRIEVE-PURCHASE. The procedures copies the work area to the File Section "generic" record area.

Step 3: The RETRIEVE-PURCHASE paragraph includes a READ statement that gets a PURCHASE-RECORD from the file that stores the database. The system loads the record into the generic record area.

Step 4: The RETRIEVE-PURCHASE paragraph copies the generic record area back into the work area for use by your own routines.

(2.11) EXAMPLE: DATA FLOW. FMS-written procedures automatically move data records between the "generic" record area (File Section) and work areas for individual record types (Working-Storage Section).

Chapter 3

THE FMS ROUTINES

Introduction

This chapter takes you through the nine FMS routines, explaining what they do and how to use them.

First, there are a few simple procedures to remember. The FMS routines are interactive: you start at the main menu by indicating the number of the routine that you want to execute, and then the FMS offers Function Choices and Data Entry Prompts, to elicit from you the information it needs to perform the desired function. As you proceed through each routine, remember:

* Press <CR> to move the cursor to the next step, or the next field to be filled.

* Press <ESC> to move backwards to the preceding step, screen or function.

* Press function key 1 <f1> to confirm entries, when the FMS prompts you to do so.

On the next page you will find a replica of the FMS main menu; this is followed by a short explanation of the purpose of each routine. The remainder of the chapter guides you through each routine, step by step.

The FMS Main Menu

Select the routine you want by typing its number in response to the File Management System's prompt.

1. DEFINE/MAINTAIN DATA ELEMENTS

Enter or change the definitions of the data elements composing your records. These definitons are stored in a "data element dictionary".

2. DISPLAY/PRINT DATA ELEMENTS

a) Display the contents of the data element dictionary -- individually or in groups.

b) Print a complete or partial list of the data element dictionary's contents.

3. DEFINE/MAINTAIN DATA BASES

a) Enter or change the following specifications:

* The name of a database structure.
* A directory to receive the FMS source code files that define the database structure.
* Whether or not the database will be stored in a single file.

b) Make a copy of a database structure under a different name.

4. DEFINE/MAINTAIN DATA BASE RECORDS

a) Enter or change the following specifications:

* A name for each record type in a database structure.
* The hierarchical structure of the database, specified by assigning a "parent" to each record type.
* The names of one or more datafiles that will store database records.

5. RESEQUENCE DATA BASE RECORDS

a) Assign new sequence numbers to record types.

6. DEFINE/MAINTAIN RECORD ELEMENTS

a) Define the structure of each record type in terms of data elements and fillers.

b) Specify the function of each data element in the record type: local (primary) key, alternate key, non-key, group header.

c) (optional) Define data-items with OCCURS or REDEFINES clauses.

7. DISPLAY/PRINT DATA BASES

a) Display or print the construction of the record types in a database structure (individually or all together).

b) Display or print the FMS's coding of the record types in a database structure (individually or all together), including the concatenated primary key area and alternate key area.

8. GENERATE DATA BASE COPY FILES

a) Initiate the automatic generation of database copy files.

9. GENERATE PROGRAM COPY FILES

a) Select the database structures and record types to be used by a particular program.

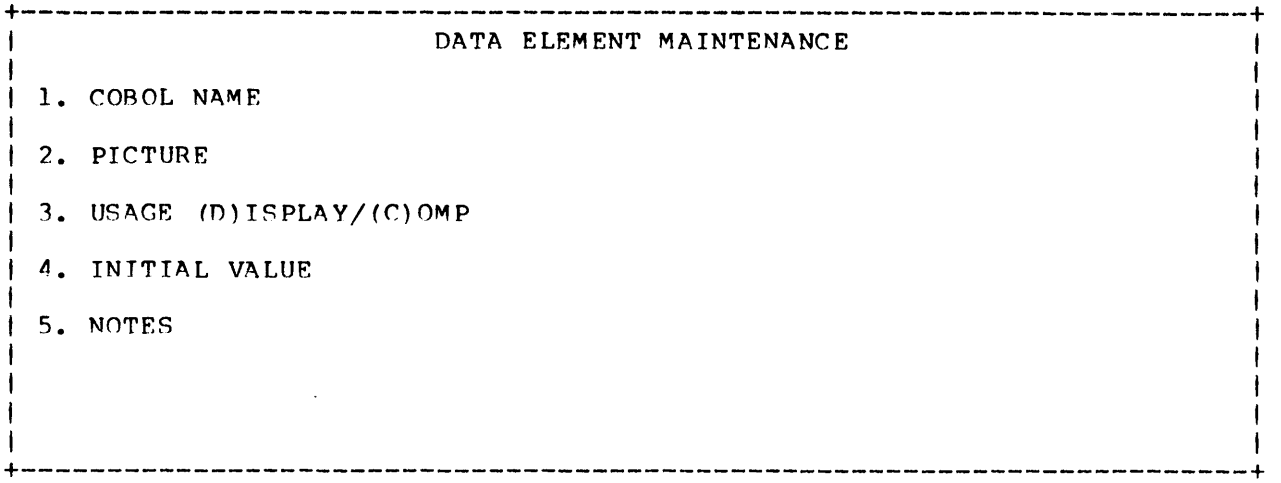
b) For each record type, choose Data Division and Procedure Division source code modules for inclusion in the program.

c) Initiate the automatic generation of program copy files.

0. RETURN TO LOGON

a) End FMS execution.

DEFINE/MAINTAIN DATA ELEMENTS Routine Menu Choice 1



This routine lets you add, modify, or delete data elements in the data element dictionary.

> NOTE: If you delete an element or modify an element name, you must later use the DEFINE/MAINTAIN RECORD ELEMENTS routine to correct any records that use the element.

Function Menu
==========

TYPE <_> TO (A)DD, (M)ODIFY, (D)ELETE ELEMENTS
(N) FOR NOTE MAINTENANCE

* Enter A to add data elements. Enter the several fields that define a data element. When you finish, the FMS automatically goes to the Note Maintenance routine (see below). After you finish note maintenance, the FMS enters the data element in the data element dictionary and clears the input fields. You may enter another data element or press <ESC> to return to the function menu.

* Enter M to modify a data element. The FMS will prompt

> ENTER COBOL NAME

Enter the name of an existing data element. The screen displays the entire data element definition, including the notes pertaining to it. Change one or more fields by typing over the existing information. After you process the last field, the FMS automatically goes to the Note Maintenance routine (see below). After you finish note maintenance, the FMS updates the data element in the data element dictionary and returns to the function menu.

* Enter D to delete data elements. The FMS prompts you to enter the name of a data element to delete. The screen displays the entire data element, including the notes pertaining to it, along with the prompt:

PRESS (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION

After you confirm or abort, the FMS will prompt you to enter another data element name for deletion. You may do so or press <ESC> to return to the function menu.

Note Maintenance
---

* At the data element maintenance function menu, you may enter N to add, delete or modify a data element's notes. (You also have these options whenever you add or modify a data element itself.) You will be prompted to enter the name of the data element you want to alter. When you do, the FMS will prompt

TYPE <_> TO (A)DD, (M)ODIFY, (D)ELETE NOTES, (P) FOR NEXT PAGE OF NOTES

* Enter A to add notes. The FMS then prompts

ENTER NOTE, (FK1) TO END

Enter one or more notes, up to 32 characters each, typing <CR> at the end of each. When you have no more notes to add, press <FK1>. The data element maintenance function menu returns.

* Enter M to modify notes. The screen will prompt

ENTER NOTE NUMBER TO MODIFY:__

Respond with the number of an existing note. The FMS then displays the text of the note and allows you to modify it. Then it returns to the same prompt, allowing you to modify another note. Press <f1> when you are finished modifying notes. The data element maintenance function menu returns.

* Enter D to delete notes. The screen will prompt you to enter the numbers, one at a time, of the notes you wish to delete. Press <f1> when you are finished deleting notes. The data element maintenance function menu returns.

* Enter P if this data element has more than 10 notes and you wish to display a note numbered above 10. The FMS displays the next page of notes.

Data Entry Prompts

1. COBOL NAME: Enter the standard COBOL data-name for the data element to be added. Permitted characters are A-Z, 0-9 and - (hyphen). Maximum length is 30 characters.

2. PICTURE: The argument for a COBOL PICTURE clause. Permitted characters are X, 9, A, S and V.

3. USAGE (D)ISPLAY/(C)OMP: If you specified X for the PICTURE clause, then DISPLAY is indicated automatically. If you specified 9, then you must indicate USAGE as D or C.

4. INITIAL VALUE: You may specify an initial value for the data element. This value is coded into the clearing record of any record type that includes the element. This step is optional and can be bypassed by pressing <CR>.

5. NOTES: Each data element in the data element dictionary can be documented by a set of notes. Each note may contain up to 32 characters. The FMS automatically numbers notes consecutively as they are entered, up to a maximum of 99 notes per data element. These notes appear in FMS printouts and displays of data element, but are not included in FMS-generated source code.

Error Messages

INVALID MAINTENANCE OPTION: The FMS received an answer it didn't expect.

COBOL NAME IS INVALID: Your data element name contains illegal characters.

<element-name> ALREADY EXISTS: A data element already exists in the data dictionary under the name you asked to be added.

<element-name> DOES NOT EXIST: No data element exists to be modified or deleted under the name you've given.

INVALID COBOL PICTURE: The picture you defined is in illegal format or contains illegal characters.

USAGE MUST BE DISPLAY OR COMP: The entry must be D or C.

DATA OVERFLOW POSSIBLE: The initial value you entered exceeds the capacity of the PICTURE.

DISPLAY/PRINT DATA ELEMENTS Routine Menu Choice 2

This routine displays the data elements you have entered in the data element dictionary, as either a complete or partial list, or one by one. It also enables you to put the list of data elements on the PASS queue for printing. (The actual printing is a separate operation.)

First, you must respond to the File Management System's prompt

ENTER COBOL NAME:

You may enter either a single data element name or the word ALL. If you enter a data element name, the FMS displays the element, using the format shown above. It then prompts you to enter another data element name. Do so, or press <ESC> to return to the main menu.

If you enter the word ALL, the FMS asks whether you wish to print the data element list, then prompts

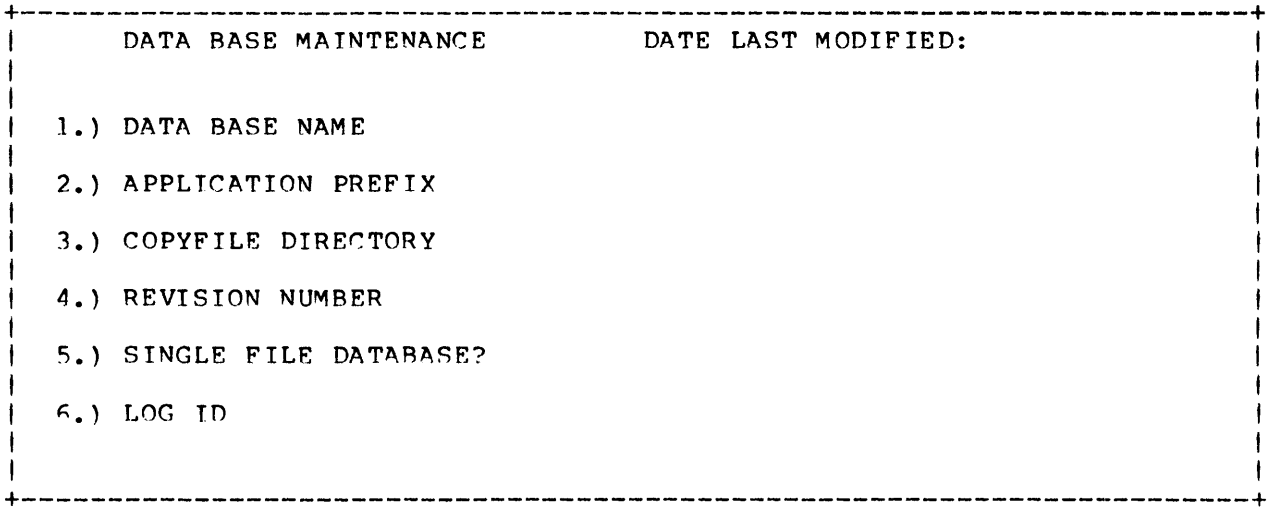
ENTER COBOL NAME TO START LISTING FROM:

You may (1) enter the name ~just prior to~ the first one (alphabetically) you want to print, or (2) press <CR> alone to list all the data elements.

If you specified printing, the FMS places the list on the PASS queue, in a file named DDELIST.

Whether you specified printing or not, the FMS displays the data element list; press <ESC> to return to the main menu.

DEFINE/MAINTAIN DATA BASES Routine Menu Choice 3



Use this routine to assign a name to a database structure, to specify the directory in which the FMS source code files for the database will be stored, and to enter information for record-logging and documentation purposes. You can also modify these specifications, delete a database structure entirely, or make a copy under a different name.

Function Menu
==========

TYPE <_> TO (A)DD, (M)ODIFY, (D)ELETE, (C)OPY A DATABASE

* Enter A to add a new database structure. Enter the several fields to define the overall specifications of a database structure. When you finish, the FMS returns to the function menu.

* Enter M to modify any of the specifications of an existing database structure. The FMS will prompt you to enter a database name. It displays the database specifications, and allows you to change one or more by typing over the existing entries. When you finish, the FMS returns to the function menu.

* Enter D to erase completely a database structure. Use it carefully; it also deletes all record types that are defined for that database structure. After you select this function and enter the name of the database structure to be deleted, the FMS displays its specifications and prompts

PRESS <FK1> TO CONFIRM, <ESC> TO ABORT

If you change your mind about deleting, press <ESC>; otherwise confirm. In either case, the FMS returns to the function menu.

* Enter C to duplicate a database structure under another name. (This is useful for making a backup copy of a database prior to introducing changes.) The FMS will prompt

ENTER NEW DATA BASE NAME

After the FMS has made the copy, press <ESC> to return to the function menu.

Data Entry Prompts

1. DATA BASE NAME: Up to 8 characters from the set A-Z and/or 0-9.

2. APPLICATION PREFIX: An optional two-character code (any two characters are permitted). The application prefix appears in record description printouts generated by DISPLAY/PRINT DATA BASES (main menu option 7), but is not used in FMS-written code.

3. COPYFILE DIRECTORY: Selects the directory that will receive the copy files generated by the GENERATE DATA BASE COPY FILES and GENERATE PROGRAM COPY FILES routines. To select the master directory, press <CR>.

4. REVISION NUMBER: Optional. Enter a two-digit number if you want to document a revision. The number will appear in record description printouts, but is not used in FMS-written code.

5. SINGLE FILE DATABASE?: Enter Y to assign all records in the database to a single file, which then automatically has the same name as the database. Enter N if you want to implement the database as two or more files. You will assign names and records to these files in the Define/Maintain Record Elements routine.

6. LOG ID: Optional. A two-character code that identifies the database for record logging purposes. The code you enter will appear in the LOG-ID data-item whenever a record in this database is deleted or modified. See Chapter 4 for details.

Error Messages

ILLEGAL CHARACTER IN DATA BASE NAME: You have entered a database name that contains characters other than A-Z or 0-9.

<database-name> DATA BASE NAME IS INVALID: You are trying to add a new database structure using a name that already exists.

<database-name> DOES NOT EXIST: You are trying to modify or delete a database structure which does not exist.

*ERROR:CHARACTER MUST BE NUMERICAL: A numeric entry must be made for REVISION NUMBER.

```
DEFINE/MAINTAIN DATA BASE RECORDS Routine                    Menu Choice 4
+-------------------------------------------------------------------------+
|                DATA BASE RECORD MAINTENANCE                             |
|                                                                         |
|                                                                         |
| SEQ  LEVEL  RECORD NAME       PARENT RECORD    COPYFILE   DATAFILE      |
|                                                                         |
|                                                                         |
|                                                                         |
|                                                                         |
+-------------------------------------------------------------------------+
```

With this option you can:

- Name each record type in a database structure.

- Create the database hierarchy by specifying a parent for each record type.

- Assign record types to data files (multiple-file databases only).

- Indicate whether or not the FMS should include record-logging code in its procedures.

If you delete a substantial number of record types or revise the database structure, you may want to use the RESEQUENCE DATA BASE RECORDS routine to reassign the sequence numbers.

The FMS first prompts

```
ENTER DATABASE NAME:
```

Enter the name of an existing database structure. The screen shown above will be displayed.

Function Menu
==========

```
TYPE <_> TO <A>DD, (M)ODIFY, (D)ELETE
         (P)ROCEED TO NEXT PAGE OF RECORDS
```

* Enter A to add record types. Enter each field at the bottom of the screen to define the overall specifications for the record type. The FMS supplies the entries for SEQ and LEVEL. When you finish defining a record type, the FMS copies your entries to the top portion of the screen and clears the input fields, allowing you to define another record type. When you are finished entering record types, press <f1> to store your work or <ESC> to cancel your work. The function menu returns.

* Enter M to modify existing record types. The FMS prompts

```
TYPE <__> SEQUENCE TO MODIFY, (A) TO STEP THRU ALL RECORDS
          (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION
```

If you type a record sequence number, that record is displayed at the bottom of the screen. You may change one or more fields by typing over the existing information. The FMS then prompts you to enter another sequence number. When you have finished modifying record types, press <f1> to store your work or <ESC> to cancel your work. The function menu returns.

> NOTE: If you change the level of a record type by specifying a different parent record, then the levels of all its dependent record types are automatically changed to reflect the new arrangement.

If you type A for (A)LL records, all records in the database are submitted for modification in order of sequence number. When you have finished, press <FK1> to store your work or <ESC> to cancel your work. The function menu returns.

* Enter D to delete an existing record. The FMS prompts

```
TYPE <__> SEQUENCE NUMBER TO DELETE
          (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION
```

The record type you select is displayed, along with all of its dependents, and the following prompt appears:

```
WARNING: THE ABOVE RECORDS WILL BE DELETED
PRESS (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION
```

If you are sure you want to delete all those record types, press <f1>. To cancel, press <ESC>. After performing the deletion, the FMS displays the names of the remaining record types and returns to the function menu.

Data Entry Prompts

SEQ: The sequence number used in the FMS record definition as a prefix to the local key area. the FMS assigns this number automatically when you add a record type to the database structure.

LEVEL: When you enter the name of the parent, the FMS automatically computes the record type's level in the database hierarchy.

RECORD NAME: A COBOL data-name up to 20 characters long.

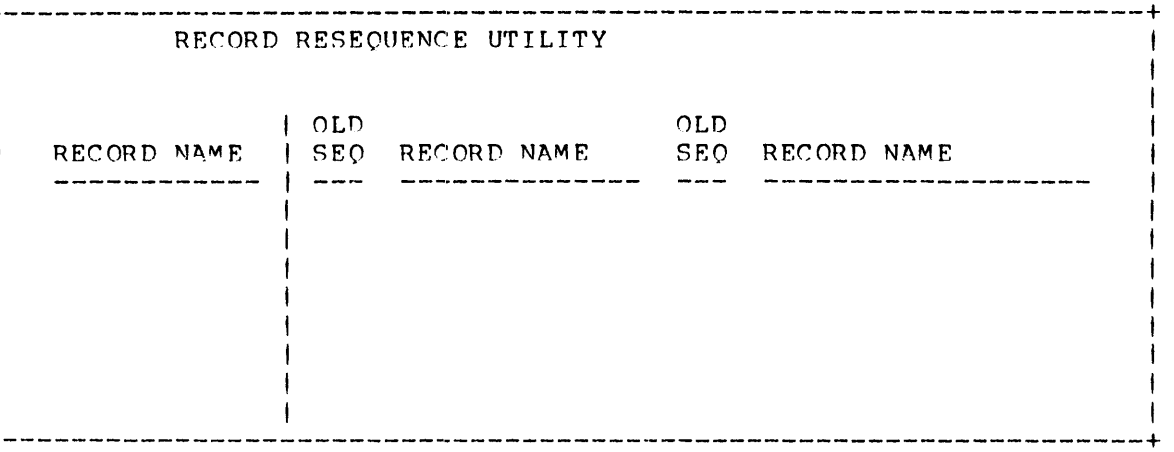
PARENT RECORD: The immediate parent of the record type you are entering. Press <CR> to accept the root record as the parent, or enter a different name.

COPYFILE: The "base" name for all "database copyfiles" the FMS will create for this record type. For each such file, the FMS adds a different filename extension to this base name.

The FMS will automatically enter a copyfile name which is the record name shortened to eight characters, and with hyphens replaced by $ signs. You can accept this copyfile name by pressing <CR>, or enter a different one.

DATAFILE: If the database is to be implemented as more than one file, enter the name of the file in which records of this type are to be stored. For single-file databases, the FMS automatically assigns a file with the same name as the database.

LOG: Type Y if records of this type are to be logged whenever they are modified or deleted by your program. See Chapter 4 for details.



There are two cases in which you may want to use this feature (but in no case are you required to):

1. You have deleted a number of record types and want to eliminate gaps in the set of sequence numbers.

2. You want to reassign sequence numbers because the parent-dependent relations between record types have changed.

Function Menu

When you select this option the FMS prompts

```
ENTER DATA BASE NAME:
```

Enter the name of the database structure for which you are resequencing the record types. The screen displays the current sequence numbers in the OLD SEQ column, the root record type and its 0 sequence number in the NEW SEQ column, and the prompt

```
ENTER OLD SEQUENCE NUMBER OF NEXT RECORD:__
PRESS (FK1) TO END, (ESC) TO ABORT.
```

* Press <f1> if you want the FMS to close the gaps between sequence numbers, and not to change the order of any record types. It will do so automatically.

* To reorder record types, enter the old sequence number of the record you want to give the 01 sequence number. If you do not want the remaining records to keep their present order, next enter the old sequence number of the record you want to give the 02 sequence number, and so on.

When the names in the NEW SEQ column, followed by those remaining in the OLD SEQ column (if any) are in the sequence you want, press <f1>. The records remaining in the OLD SEQ column will be resequenced automatically.

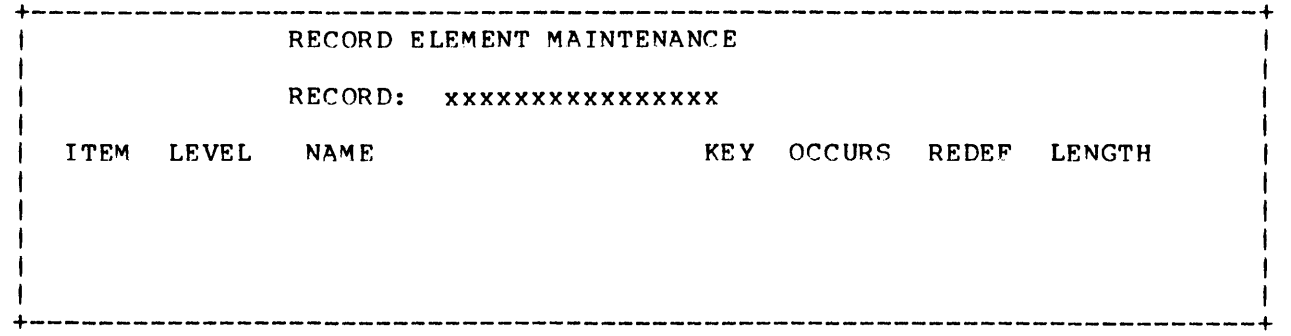
When resequencing is complete, the FMS will prompt you to enter another database name for resequencing. To return to the main menu, press <ESC>.

Error Messages

*ERROR: CHARACTER MUST BE NUMERIC: You entered a non-numeric character or CR instead of a sequence number. Try again.

RECORD DOES NOT EXIST: You entered a number to be resequenced, which does not exist in the OLD SEQ.

DEFINE/MAINTAIN RECORD ELEMENTS Routine Menu Choice 6



Use this routine to define record layouts in terms of data elements selected from the data element dictionary. As you select data elements, the FMS prompts you to supply additional information to produce complete data-item entries. The additional information includes a COBOL level number, and, optionally, an OCCURS or REDEFINES clause.

The FMS first prompts

ENTER DATA BASE NAME __

After you enter the name of an existing database structure, the FMS prompts

ENTER RECORD NAME __
(ALL) TO STEP THROUGH ALL RECORDS

Function Menu

After you enter either ALL or a record name the FMS displays the screen shown above, with the database and record names at the top and the elements (if any) currently assigned to the record type. The FMS prompts

TYPE <_> TO (A)DD, (M)ODIFY, (D)ELETE, (I)NSERT
(P) FOR NEXT PAGE OF ELEMENTS, (N) FOR NEXT RECORD

* Enter A to define new data-items in a record type for the first time, or to add elements beyond the last existing one. Enter the fields at the bottom of the screen to define a single data-item. When you have finished, the FMS copies the item to the top portion of the screen and clears the input fields, allowing you to define another new data-item. You may do so, or press <ESC> to return to the function menu.

* Enter M to modify existing data-items. The FMS prompts

TYPE <__> ITEM TO MODIFY, (A) TO STEP THROUGH ALL ELEMENTS
PRESS (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION

Enter either the number of an existing data-item entry, or the letter A to step through all the items defined in this record type. For each data-item the FMS displays, you may change one or more fields by typing over the existing information. When you finish modifying an item, you may enter another number to continue modifying data-items. Otherwise, press <f1> to enter all modified items into the record definition or press <ESC> to cancel your work. The function menu returns.

* Enter D to delete previously-entered data-items. The FMS prompts

```
TYPE <  > ITEM TO DELETE
PRESS (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION
```

Enter the numbers, one at a time, of the items to be deleted. When you have finished, press <f1> to confirm the deletions or press <ESC> to cancel. The function menu returns.

* Enter I to insert new data-item entries between two existing ones. the FMS prompts

```
TYPE <  > ITEM TO INSERT AFTER
PRESS (FK1) TO CONFIRM, (ESC) TO ABORT FUNCTION
```

Enter the number of the data-item after which the new one is to be inserted; OR, press <CR> to insert a new item before the first one. You construct the new data-items at the bottom of the screen (just as in the ADD function). When you finish a data-item, the FMS clears the input fields. You may enter another data-item, press <f1> to confirm the insertions, or press <ESC> to cancel them. Existing items subsequent to the inserted items are renumbered automatically, and the function menu returns.

Data Entry Prompts

ITEM: The FMS automatically assigns an item number to each data-item added. Item numbers are used only in screen operations; they are not used in FMS-written source code.

LEVEL: This is the COBOL level of a Data Description Entry for the data element. Use any number from 03 to 49.

NAME: Enter either the name of a data element taken from the data element dictionary, or the word FILLER. If you enter FILLER, the cursor moves automatically to the "length" field.

KEY: Your choices are Y, A, or N. Y means this element is part of the local key for this record. A means this data element is part of the record's alternate key; and N means the element is not part of a key (you can also press <CR> for No).

The primary key must be the first element (or elements) listed and the alternate key must be the second.

> NOTE: You may choose to have a key consist of more than one data-item, or of a group of data-items. If you do, the items must be listed consecutively; if the key is a group, make sure the group-name item, as well as each component item, is labelled as a key.

OCCURS: Optional; used to specify a table or array. By entering a number here you introduce a COBOL OCCURS clause with that number as its argument. For example, if an element is defined in the data element dictionary as

```
AA      PIC  X(03)
```

if you specify 05 for level and 10 for OCCURS, the data element AA will appear in this record as a data description entry that looks like this:

```
05      AA          PIC X(03)          OCCURS 10 TIMES
```

> Note: The OCCURS clause can be applied to the highest level of a group item, or to the lowest level, but not to intermediate levels. In other words, you can make a whole group, or an elementary item, repeat itself, but not a subgroup.

REDEF: Optional. Enter Y to make the current data element redefine the previous data element. You cannot give an item both the OCCURS and the REDEFINES clause; nor may you redefine an item containing an OCCURS or REDEFINES clause.

LENGTH: If the item is a data element, the FMS enters the length of the element automatically. If you named the data-item FILLER, you must specify its length.

Error Messages

<your entry> DOES NOT EXIST IN DATA DICTIONARY: The data element you are trying to add is not listed in the data element dictionary.

ELEMENT TO MODIFY IS NOT ON THIS PAGE: The item number of the data element you have given to be modified is not on the page displayed. To go to another page, enter P.

ANSWER MUST BE Y/N/A: In the key field you must enter (Y)ES, (N)O, or (A)LTERNATE.

ELEMENT DOES NOT EXIST: The data element you are trying to modify or delete is not included in the record.

LEVEL MUST BE BETWEEN 03 & 49: The COBOL level assigned to a data element must be between 03 and 49, inclusive.

This routine allows you to display and/or print an existing database structure. The display or printout lists the names of record types and files used in the database structure, along with the layout of each record type.

First enter the name of a database you defined in main menu option 3. The FMS will display the date it was defined or last modified, and move the cursor to the RECORD field.

Enter ALL to process all record types in the database structure; otherwise enter the name of a single record type. The FMS will complete the rest of the fields shown above and prompt

ENTER (1) DISPLAY, (2) PRINT, (3) BOTH 1 & 2

If you choose printing, the FMS asks a question concerning the format in which records are to be printed:

PRINT WORKSHEET (Y/N)?

Examples of both formats (programmer's worksheet and source code summary) are provided below. The programmer's worksheet lists the record types as they were defined in the "Define/Maintain Data Base Records" routine. The source code summary lists the actual record structures the FMS will code, including extra fields in the key portion of the record definition.

If you specify printing, the listing is directed to file <database-name>.CH, which is placed on the PASS queue.

Error Messages

CALCULATING TOTAL KEY LENGTH....PLEASE WAIT: Self-explanatory.

<your entry> DOES NOT EXIST: the FMS does not recognize the name by which you have asked for a record.

INVALID OUTPUT REQUESTED: An invalid response has been given to a multiple-choice prompt.

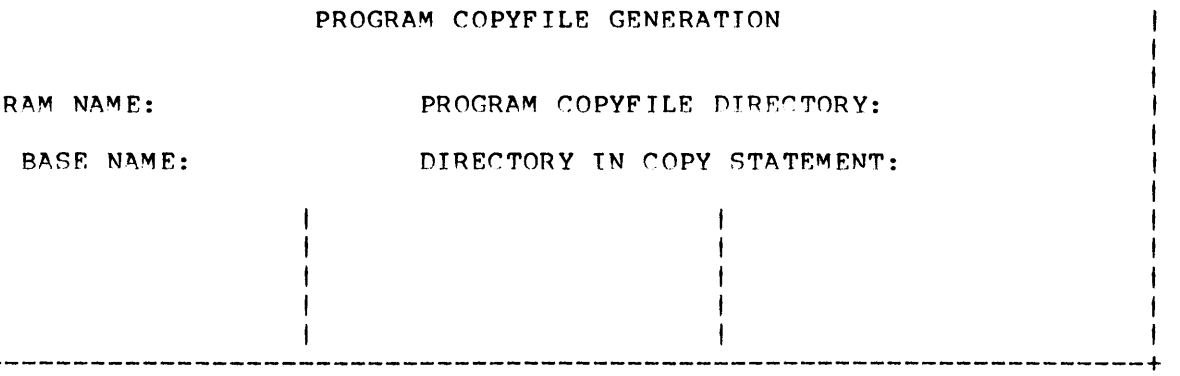
GENERATE DATA BASE COPYFILES Routine — Menu Choice 8

When the data base definition is complete, use this routine to generate a set of COPY files for incorporation into COBOL programs that will access the database.

These COPY files contain all the COBOL source code that is needed to define the database structure, and a set of procedures to access each record type. Appendix B lists the names and contents of all database copyfiles.

Enter a database name; the FMS will then produce all relevant database copyfiles. These copyfiles, as well as a documentation file <database-name>.$D, are placed in the directory you named in the "Define/Maintain Data Bases" routine. The FMS prompts you to enter another database name. You may do so, or press <ESC> to return to the main menu.

GENERATE PROGRAM COPYFILES Routine Menu Choice 9



With this routine you tell the FMS which database structures and which records a program is using and how it is using them, and you generate program copyfiles.

Data Entry Prompts

First you have to give the FMS some information.

PROGRAM NAME: Enter the name of a COBOL program.

PROGRAM COPYFILE DIRECTORY: Optional. You may enter the name of a directory to receive the FMS's Program Copyfiles; if you enter <CR>, the FMS will assign them to the directory specified in main menu option 3.

DATA BASE NAME: The name of a database to be processed by the program.

DIRECTORY IN COPY STATEMENT: the FMS automatically displays the directory you named in the "Define/Maintain Data Bases" routine to store the database copyfiles.

At this point the FMS displays a numbered listing of the record types in the database structure and prompts

```
ENTER RECORD NUMBER:_____
PRESS (FK1) TO END, (ESC) TO RE-ENTER DATA BASE.
```

Enter the number of a record type to be used by your program. (Pressing <ESC> returns you to the database name prompt.) The FMS then prompts

```
ENTER (Y/N) FOR CLEARING RECORD
PRESS (ESC) TO RE-ENTER RECORD NUMBER
```

Enter Y if your program needs a clearing record for the record addressed. Enter N if no clearing record is required. Pressing <ESC> returns you to the previous prompt.

The FMS prompts

```
ENTER (R) FOR READ ONLY, (W) FOR READ/WRITE, (D) FOR READ/WRITE/DELETE
PRESS (ESC) TO RE-ENTER RECORD NUMBER
```

Enter R to include in the program copyfile a COPY statement that incorporates FMS procedures to read records of the specified type.

Enter W to include in the program copyfile a COPY statement that incorporates FMS procedures to read, write, and rewrite records of the specified type.

Enter D to include in the program copyfile a COPY statement that incorporates FMS procedures to read, write, rewrite, and delete records of the specified type.

Pressing <ESC> cancels processing of the record type and returns you to the ENTER RECORD NUMBER prompt.

After you have completed these entries, the FMS returns to the ENTER RECORD NUMBER prompt, allowing you to specify another record type from the same database structure. When you have finished specifying record types from this database, press <f1>. The FMS then returns to the database name prompt, allowing you to specify additional databases to be processed by the program.

Chapter Four

Programming with the FMS

This chapter explains how to use FMS-written code. First, we discuss the few simple steps required to include the code in a COBOL program. Then, we discuss the ways in which you use the File Management System's data definitions and database-access procedures.

Incorporating FMS Code in a Program

The FMS's "Generate Program Copyfiles" routine has stored in "program copyfiles" the COPY statements that will incorporate FMS Data and Procedure Division code into the program at compilation time. You can insert the contents of these files into your program using a simple CRT/EDIT or IC/EDIT command:

IC/EDIT: Issue the COPy command and respond to the prompts as follows:

AFTER: The number of a line in the appropriate Division or Section after which the code should be inserted.

FROM FILE: <program-copyfile-name>

CRT/EDIT: Move the cursor to a line in the appropriate Division or Section after which the code should be inserted.

Issue the command: FI<program-copyfile-name>

The following table lists the names of the program copyfiles and the program locations at which their contents are to be inserted:

Program copyfile name	To be inserted in
<database-name>.$S	Environment Division, Input-Output Section
<database-name>.$F	Data Division, File Section
<database-name>.$W	Data Division, Working-Storage Section
<database-name>.$P	Procedure Division

Appendix B describes the contents of these files.

Most of the Data Division code you include according to these instructions has already been described in detail: discriptions of "generic" RECORD AREAs and of Working-Storage "work areas" for each record type. The FMS also includes a few extra elementary Working-Storage data-items for general use. In addition, you need to define several data-items yourself.

General Purpose Working-Storage Items

The FMS includes in the program copyfile <program-name.SW> a set of data-items used by all FMS procedures, no matter what database is being processed. The usage of these data-items is as follows:

DISPLAY-FILE-ID PIC X(24)

Holds the name of the data file which stores the record most recently accessed by the program. This information is useful for logging purposes and for inclusion in data-entry screen formats.

ERROR-CODE PIC 9(02)

This is the FMS's own "file status" data-item. You should interrogate ERROR-CODE after most calls to FMS procedures. The procedures use this item to return a value that indicates the outcome of the input/output operation:

* A zero value means that the operation was successfully completed.
* A non-zero value indicates an error condition.

The meanings of the ERROR-CODE values are listed in the "FMS Error Codes" section of this chapter.

FILE-STATUS-CODE PIC X(01)

Specified in FMS-written SELECT statements as the official "file status" data-item for all FMS data files. The FMS's RETRIEVE, GET-NEXT, and DELETE procedures use this field to determine which value should be loaded into ERROR-CODE. You need not address this field with your own routines.

LOG-RECORD-TYPE PIC 9(02) COMP

Contains the sequence number of the last record accessed by an FMS procedure. This information is primarily of interest for record logging.

LOG-ID PIC X(02)

The two-character code that you specified in the "Define/Maintain Databases" routine to identify the database for logging purposes.

REREAD-COUNT PIC 9(03)

Used by DELETE-<record-name> procedures that must repeatedly attempt to delete a dependent record that has been LOCKed by another program.

Programmer-Defined Working-Storage Items

Some of the FMS's optional features require additional Working-Storage data-items to support their functionality:

Alternate Key Access

<filename>-ALTERNATE-KEY-FLAG PIC 9

You must define a data-item of the above form for each data file that stores FMS database records accessed with an alternate key.

Record Logging

LOG-ENTRY-TYPE PIC X
LOG-DATA-AREA <programmer-defined PICTURE>

You must define these two items, whether your program does any record logging or not.

Just before an FMS-written procedure issues a "PERFORM LOGGING-ROUTINE" statement, it moves a character to the LOG-ENTRY-TYPE field to indicate the type of logging transaction:

Character	Type Of Transaction
J	new record created by INSERT-<record-name> procedure
B	old record overwritten by REPLACE-<record-name> procedure
A	new record written by REPLACE-<record-name> procedure
D	old record (plus dependents) deleted by DELETE-<record-name> procedure

Just before an FMS procedure issues a "PERFORM LOGGING-ROUTINE" statement, it MOVEs the record to be logged to LOG-DATA-AREA.

The Procedure Division of an FMS-Based Program

In the Procedure Divison of your program, you must (1) write statements that OPEN and CLOSE the file(s) that store the FMS databases, and (2) write routines that call FMS procedures and test the results.

Opening and Closing the Database Files

The syntax of an OPEN statement is:

```
                              {  INPUT    }
                              {  OUTPUT   }
        OPEN  [ EXCLUSIVE ]   {  I-O      } <database-file>
                              {  EXTEND   }
```

The syntax of a CLOSE statement is:

```
        CLOSE  <database-file>  [ WITH LOCK ]
```

Choose the forms of these statements appropriate for your program. You may also include several database filenames in a single OPEN or CLOSE statement. See the ~Interactive COBOL Programmer's Reference~ for details.

Writing Routines that Call and Test FMS Procedures

In this section, we discuss the principle task in adapting a program to use FMS database structures: writing routines that call the database-access procedures the FMS has written. These procedures are your tools for performing standard input/output operations on a record-by-record basis.

In general, use the following steps to access a record in the database:

1. Load values into the key elements defined in the work area defined in Working-Storage for the record type. You must include all the local keys that compose the concatenated RECORD KEY that the COBOL interpreter actually uses to access the database: the local key of the record type to be processed, the local key of its parent, the local key of its parent's parent, and so on back to the root of the database tree.

Typically, you'll use the MOVE and/or ACCEPT verb to load the key values. For processing of multiple dependent records, you may "hold constant" the key values of the parent(s) while varying the local key value of the dependent.

2. Write a statement that PERFORMs an FMS-written procedure. The procedure moves the contents of the work area (Working-Storage Section) to the generic record area (File Section), executes an input/output statement, and (if appropriate) moves newly-retrieved data from the generic area back into the proper work area.

3. Test the value of ERROR-CODE, the FMS's own "file status" item, to determine whether the input/output operation was successful, and branch accordingly.

These three steps are treated in the following sections.

Loading the Key Elements

Most FMS procedures perform a keyed access of the database file to locate or create a particular record. Before you call one of these procedures, you must make sure that the key fields of the work area are loaded with the values that identify the record. Here are a few important points:

* Never address the RECORD KEY in the File Section RECORD AREA directly. FMS procedures take care of this. Rather, you should always use the work areas the FMS defines for the individual record types in Working-Storage.

* You must ensure that the entire set of local key elements have the proper values. If, for instance, you plan to WRITE a level-3 record, you must load the extra fields that the FMS codes in the work area to hold the level-2, level-1, and level-0 key values. Figure 4.01 illustrates the key-loading required before you "PERFORM INSERT-PAYMENT" to add a PAYMENT-RECORD to the purchses/payments database.

Often, you won't need to load values for all local keys. If you want to read several level-3 records of the same type, you need to load the level-2, level-1, and level-0 fields only for the first RETRIEVE-<record-name> access. Thereafter, you just change the level-3 local key value while holding the other keys constant.

* The FMS uses the same data-name to code both a parent's local key and the corresponding "extra" field in a dependent's key area. To prevent ambiguity, you must qualify all references to key elements -- both primary and alternate.

Clearing Records

It is often desirable to "blank out" the non-key elements of a record type's work area before loading it with values. For each record type, the FMS creates a ~clearing record~ with the same data element structure. This area, named WSC-<record-name>, is initialized with "VALUE IS" clauses to SPACES in non-key alphanumeric elements and to ZERO in non-key numeric elements. To effectively clear the non-key portion of the work area for any record type, simply overwrite the work area with the clearing record:

```
MOVE  WSC-<record-name>-RECORD  TO  <record-name>-RECORD
```

NOTES:

* Never initialize a record type's work area by a "MOVE SPACES TO <record-name>-RECORD or a "MOVE ZERO TO <record-name>-RECORD. This will destroy the sequence number prefixes that establish the record type's position in the database structure.

* One programming strategy is never to change the contents of the clearing record. In this case, the contents of the key portion of the clearing record are unpredictable. Hence, you must use the clearing record ~before~ you load primary or alternate key values.

* Another programming strategy is particularly useful in processing several dependent records with the same parent: Load the clearing record with the local key to the parent record, the local key to the parent's parent, etc. In this case, moving the clearing record to the work area both blanks the non-key fields and establishes all key values except that of the dependent's local key.

The FMS Database-Access Procedures -- Programmer's Reference

The following sections describe the set of database-access procedures the FMS provides for your use. The examples used, including page and line references -- such as, "(p10 / 4)" -- are taken from the compilation listing of the demonstration program, DEM$FMS, included on the File Management System release medium. For instructions on producing this compilation listing, see Appendix A.

First, a table is provided as a quick reference:

Procedure name	Function
DELETE-<record-name>	Delete a record
GET-NEXT-<record-name>	Read the next record in a sequence of records
INSERT-<record-name>	Create a new record
LOCK-NEXT-<database-name>-RECORD	Set flag to lock next record read
REPLACE-<record-name>	Rewrite an existing record
RETRIEVE-<record-name>	Read an existing record
SET-<file-name>-ALT-KEY-1-OFF	Set flag to start ALTERNATE KEY reads
SET-<file-name>-ALT-KEY-1-ON	Set flag to end ALTERNATE KEY reads
UNLOCK-<database-name>-RECORDS	Unlock all records in a database

Deletes the record of type <record-name> whose key is stored in the <record-name>-RECORD work area, along with all of its dependent records (if any).

Records are deleted with Interactive COBOL DELETE statements, which cause the record to be "logically" deleted (that is, flagged as deleted) rather than physically deleted. Such records may be recoved using UNDELETE statements. After a record is deleted, you may write a new record using the same RECORD KEY. The new record physically overwrites the old record in disk storage.

Record Logging

The procedure LOGGING-ROUTINE is performed each time a record and its dependents are deleted. Before this routine is called:

* The data-item LOG-ENTRY-TYPE is loaded with "D".
* A copy of the record is moved to LOG-DATA-AREA.
* The data-item LOG-RECORD-TYPE is loaded with the sequence number of the deleted record.

Incomplete Record Deletion

If one of the record's dependents is LOCKed by another program during the performance of DELETE-<record-name>, up to ten retries are attempted. If the LOCKed dependent becomes UNLOCKed before all the retries are exhausted, the deletion operation continues. If the dependent record remains LOCKed, the deletion operation aborts, with the following effects:

* The record of type <record-name> is restored.
* Dependent records preceding the LOCKed record remain deleted.
* The LOCKed record and all dependent records following it remain intact.
* The value 7 is moved to ERROR-CODE.

Example

```
START-A-CUSTOMER.                                          (p10 / 4)
    MOVE WSC-CUSTOMER-RECORD TO CUSTOMER-RECORD.
        .
        .
    ACCEPT CUST-ID-FIELD ON ESCAPE GO TO MAIN-LINE.
```

```
DELETE-A-CUSTOMER-RECORD.                                    (p11 / 2)
    PERFORM RETRIEVE-CUSTOMER.
        .
        .
    PERFORM VERIFY-CYCLE.
    IF Y-N = "Y"
        PERFORM DELETE-CUSTOMER
        IF ERROR-CODE NOT = ZERO STOP RUN.


DELETE-CUSTOMER.                                             (p18 / 4C)
    MOVE ZERO TO ERROR-CODE.
    MOVE CUSTOMER-RECORD TO PURPAYFL-RECORD.
    MOVE PURPAYFL-KEY TO PURPAY-BUFFER.
    MOVE "PURPAYFL" TO DISPLAY-FILE-ID.
    START PURPAYFL KEY = PURPAYFL-KEY
       INVALID KEY MOVE "1" TO ERROR-CODE.
    IF ERROR-CODE IS EQUAL TO ZERO
      MOVE HIGH-VALUE TO PREFIX-LEVEL-1 OF PURPAY-BUFFER
      PERFORM DELETE-PURPAYFL-NEXT-RECORD
        UNTIL PURPAYFL-KEY > PURPAY-BUFFER.
    IF ERROR-CODE IS EQUAL TO "7"
      MOVE CUSTOMER-RECORD TO PURPAYFL-RECORD
      PERFORM UNDELETE-PURPAYFL-RECORD.
```

Used repetitively to read a sequence of records of the same type.

This routine uses the READ NEXT RECORD statement, which retrieves records according to their physical order in disk storage. Records are ordered according to their local key values. The order is "ascending ASCII", essentially an extension of alphabetical order.

Notes on Key Loading

You need to load key values only once for each series of records to be retrieved with GET-NEXT-<record-name>. After you have retrieved the first record in the series, subsequent records of the same type are available with additional calls to the procedure.

To retrieve the first of a series of dependent records:

1. Load the key values that identify the parent record.

2. (optional) Issue a RETRIEVE-<record-name> statement to verify that the parent record exists.

3. (optional) Issue a LOCK-NEXT-<database-name>-RECORD statement. See "Record Locking" below.

4. Issue a PERFORM GET-NEXT-<record-name> statement.

5. Check the ERROR-CODE item.

To retrieve the first of a series of root records, simply use the RETRIEVE-<record-name> procedure, as outlined in steps 1 and 2 above. GET-NEXT-<record-name> fails to locate a record if:

* There are no more records of the same type that are dependent on the same parent record.

* It reaches the end of the database file.

Record Locking

Before any call to GET-NEXT-<record-name>, you may PERFORM the FMS procedure LOCK-NEXT-<database-name>-RECORD. This causes the READ statement to "lock" the record, ensuring that your program has exclusive rights to the record. No other program may alter this record until you issue a call to the UNLOCK-<database-name>-RECORDS procedure.

NOTE: GET-NEXT-<record-name> cannot be used to retrieve keyless records. Since there may be only one record of each keyless record type, there is no "next keyless record".

Example:

```
DISPLAY-PURCHASE-RECORDS.                                    (p16 / 28)
                  .
                  .
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PURCHASE-RECORD.
    MOVE LOW-VALUES TO PO-NUMBER OF PURCHASE-RECORD.
    PERFORM DISPLAY-A-PURCHASE UNTIL ERROR-CODE NOT = ZERO.

DISPLAY-A-PURCHASE.                                          (p16 / 39)
    PERFORM GET-NEXT-PURCHASE.
    IF ERROR-CODE = 0
      MOVE SPACES TO LINE-BUFFER
      MOVE PO-NUMBER OF PURCHASE-RECORD TO PURCHASE-COL-05
        .
        .


GET-NEXT-PURCHASE.                                           (p19 / 10C)
    MOVE PURPAYFL-KEY TO PURPAY-BUFFER.
    MOVE PURCHASE-RECORD TO PURPAYFL-RECORD.
    MOVE ZERO TO ERROR-CODE.
    IF LEVEL-1 OF PURPAYFL IS NOT EQUAL TO
       LEVEL-1 OF PURPAY-BUFFER
            PERFORM START-PURPAYFL.
    IF ERROR-CODE IS EQUAL TO ZERO
       PERFORM RETRIEVE-NEXT-PURPAYFL-LEVEL-1.
    IF ERROR-CODE IS EQUAL TO ZERO
       MOVE PURPAYFL-RECORD TO PURCHASE-RECORD.
```

Write a new record of type <record-name> into disk storage, using the contents of the <record-name>-RECORD work area.

Record Logging

If you have specified logging for the <record-name> type, the procedure LOGGING-ROUTINE is performed each time a record is created. The following MOVEs take place:

* The data-item LOG-ENTRY-TYPE is loaded with "I".
* A copy of the record is moved to LOG-DATA-AREA.
* The data-item LOG-RECORD-TYPE is loaded with the sequence number of the created record.

Example

```
START-A-PURCHASE.                                          (p12 / 16)
    MOVE WSC-PURCHASE-RECORD TO PURCHASE-RECORD.
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PURCHASE-RECORD.
        .
        .
    ACCEPT PUR-PO-NUMBER-FIELD
          ON ESCAPE GO TO DETERMINE-A-PURCHASE-CUSTOMER.


ADD-A-PURCHASE-RECORD.                                     (p13 / 3)
                  .
                  .
    ACCEPT PUR-NON-KEY-FIELDS
        ON ESCAPE GO TO START-A-PURCHASE.
                  .
                  .
    PERFORM VERIFY-CYCLE.
    IF Y-N = "Y"
       PERFORM INSERT-PURCHASE
       IF ERROR-CODE NOT = ZERO STOP RUN.


INSERT-PURCHASE.                                           (p19 /22C)
    MOVE PURCHASE-RECORD TO PURPAYFL-RECORD.
    PERFORM WRITE-PURPAYFL.
```

LOCK-NEXT-<database-name>-RECORD

LOCK-NEXT-<database-name>-RECORD

Sets the flag <database-name>-LOCK-FLAG to cause the program to LOCK the next record read from <database-name>. You may use this procedure immediately before calling the RETRIEVE-<record-name> or GET-NEXT-<record-name> procedures. The record remains locked until execution of the UNLOCK-<database-name>-RECORDS procedure.

The RETRIEVE-<record-name> and GET-NEXT-<record-name> procedures clear this flag automatically when they finished execution.

Example

```
LOCK-NEXT-PURPAY-RECORD.                                    (p24 / 3C)
   MOVE "L" TO PURPAY-LOCK-FLAG.
```

Rewrites an existing record of type <record-name> to disk storage, using the contents of the <record-name>-RECORD work area.

Record Logging

If you have specified logging for the <record-name> type, the procedure LOGGING-ROUTINE is performed twice just before a record is rewritten: first to log the version of the record read from disk, and then to log the updated record that will overwrite it. The following MOVEs take place:

* The data-item LOG-ENTRY-TYPE is loaded with "B".
* A copy of the RECORD AREA, containing the version of the record read from disk, is moved to LOG-DATA-AREA.
* The data-item LOG-RECORD-TYPE is loaded with the sequence number of <record-name>.
* LOGGING-ROUTINE is PERFORMed.
* The data-item LOG-ENTRY-TYPE is loaded with "A".
* A copy of the <record-name-RECORD> work are, containing the updated version of the record, is moved to LOG-DATA-AREA.
* LOGGING-ROUTINE is PERFORMed again.

Example

```
START-A-PAYMENT.                                          (p14 /16)
        .
        .
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PAYMENT-RECORD.
    MOVE SPACES TO INVOICE-NUMBER OF PAYMENT-RECORD.
        .
    ACCEPT PAY-INV-NUMBER-FIELD
        ON ESCAPE GO TO DETERMINE-A-PAYMENT-CUSTOMER.


CHANGE-A-PAYMENT-RECORD.                                  (p15 / 14)
        .
        .
    ACCEPT PAY-NON-KEY-FIELDS
        ON ESCAPE GO TO START-A-PAYMENT.
        .
    PERFORM VERIFY-CYCLE.
    IF Y-N = "Y"
        PERFORM REPLACE-PAYMENT
        IF ERROR-CODE NOT = ZERO STOP RUN.
```

```
REPLACE-PAYMENT.                                          (p20 / 32C)
   MOVE "LG" TO LOG-ID.
   MOVE ZERO TO ERROR-CODE.
   MOVE PAYMENT-RECORD TO PURPAYFL-RECORD.
   MOVE "PURPAYFL" TO DISPLAY-FILE-ID
   READ PURPAYFL
       INVALID KEY MOVE "1" TO ERROR-CODE.
   IF ERROR-CODE IS EQUAL TO ZERO
      MOVE "B" TO LOG-ENTRY-TYPE
      MOVE PURPAYFL-RECORD TO LOG-DATA-AREA
      MOVE 3 TO LOG-RECORD-TYPE
      MOVE "LOG-FILE" TO DISPLAY-FILE-ID
      PERFORM LOGGING-ROUTINE
      MOVE "A" TO LOG-ENTRY-TYPE
      MOVE PAYMENT-RECORD TO LOG-DATA-AREA
      MOVE "LOG-FILE" TO DISPLAY-FILE-ID
      PERFORM LOGGING-ROUTINE
      MOVE PAYMENT-RECORD TO PURPAYFL-RECORD
      PERFORM REWRITE-PURPAYFL.
```

```
================================================================================
RETRIEVE-<record-name>                                  RETRIEVE-<record-name>
================================================================================
```

Reads a record of type <record-name> from disk storage into the <record-name>-RECORD work area.

Example

```
START-A-PAYMENT.                                                    (p14 / 16)
        .
    MOVE WSC-PAYMENT-RECORD TO PAYMENT-RECORD.
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PAYMENT-RECORD.
        .
        .
    ACCEPT PAY-INV-NUMBER-FIELD
        .


CHANGE-A-PAYMENT-RECORD.                                            (p15 / 14)
    PERFORM RETRIEVE-PAYMENT.
    IF ERROR-CODE NOT = ZERO GO TO START-A-PAYMENT.
    DISPLAY PAY-KEY-FIELDS, PAY-NON-KEY-FIELDS.
        .
        .


RETRIEVE-PAYMENT.                                                   (p20 / 4C)
   MOVE PAYMENT-RECORD TO PURPAYFL-RECORD.
   PERFORM READ-PURPAYFL.
   IF ERROR-CODE IS EQUAL TO ZERO
      MOVE PURPAYFL-RECORD TO PAYMENT-RECORD.
```

SET-<file-name>-ALT-KEY-1-OFF

Puts the program in "primary key access" mode. Until you issue the complementary statement SET-<file-name>-ALTERNATE-KEY-1-ON, all RETRIEVE-<record-name> routines will READ a record according to the primary key rather than the alternate key.

SET-<file-name>-ALT-KEY-1-OFF accomplishes this task by setting the value of a "flag" data-item named <database-name>-ALTERNATE-KEY-FLAG to 0.

Example

```
START-A-PAYMENT.                                              (p14 / 16)
    PERFORM SET-PURPAYFL-ALT-KEY-1-OFF.
    MOVE WSC-PAYMENT-RECORD TO PAYMENT-RECORD.
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PAYMENT-RECORD.
        .
        .
    IF INVOICE-NUMBER OF PAYMENT-RECORD = SPACES
        AND FUNCTION-CHOICE NOT = 1
                PERFORM SET-PURPAYFL-ALT-KEY-1-ON
                ACCEPT PAY-PO-NUMBER-FIELD
                   ON ESCAPE GO TO START-A-PAYMENT.

SET-PURPAYFL-ALT-KEY-1-OFF.                                   (p22 / 46C)
    MOVE 0 TO PURPAYFL-ALTERNATE-KEY-FLAG.
```

Puts the program in "alternate key access" mode. Until you issue the complementary statement SET-<file-name>-ALTERNATE-KEY-1-OFF, all RETRIEVE-<record-name> routines will READ a record according to the alternate key rather than the primary key.

SET-<file-name>-ALT-KEY-1-ON accomplishes this task by setting the value of a "flag" data-item named <database-name>-ALTERNATE-KEY-FLAG to 1.

Example

```
START-A-PAYMENT.                                              (p14 / 16)
    MOVE WSC-CUSTOMER-RECORD TO CUSTOMER-RECORD.
    MOVE WSC-PAYMENT-RECORD TO PAYMENT-RECORD.
    MOVE ID-NUMBER OF CUSTOMER-RECORD TO
         ID-NUMBER OF PAYMENT-RECORD.
        .
        .
    ACCEPT PAY-INV-NUMBER-FIELD
        ON ESCAPE GO TO DETERMINE-A-PAYMENT-CUSTOMER.
    IF INVOICE-NUMBER OF PAYMENT-RECORD = SPACES
        AND FUNCTION-CHOICE NOT = 1
            PERFORM SET-PURPAYFL-ALT-KEY-1-ON



SET-PURPAYFL-ALT-KEY-1-ON.                                    (p22 / 43C)
    MOVE 1 TO PURPAYFL-ALTERNATE-KEY-FLAG.
```

Unlocks all records in the file(s) that store the <database-name> records.

Example

```
UNLOCK-PURPAY-RECORDS.                                    (p24 / 6C)
    UNLOCK PURPAYFL RECORDS.
```

FMS Error Codes

As a matter of course, you should follow each call to an FMS-wriiten procedure with a check on the success or failure of the input/output task performed. Exceptions: The following procedures don't perform any input/output (they merely set flags), so you need not perform a check after calling these procedures.

```
LOCK-NEXT-<database-name>-RECORD
SET-<file-name>-ALT-KEY-1-OFF
SET-<file-name>-ALT-KEY-1-ON
```

The data-item ERROR-CODE is your means of monitoring these input/output operations. FMS procedures automatically load a one-digit number into ERROR-CODE that indicates either success or the particular type of failure. Thus, this data-item performs very much the same function as the FILE-STATUS item defined for each COBOL data file.

For efficiency of execution, test for literal values of ERROR-CODE rather than for numeric values. For instance, the statement

```
IF ERROR-CODE = "0" PERFORM KEEP-ON-GOING
```

is preferable to the statement

```
IF ERROR-CODE = 0 PERFORM KEEP-ON-GOING.
```

These are the FMS error codes:

Error Code	Meaning
0	Input/output operation was successful.
1	INVALID KEY condition.
2	(only if LOCK-NEXT-<database-name>-RECORD has not been used)
3	(only if LOCK-NEXT-<database-name>-RECORD has been used) The record retrieved by GET-NEXT-<record-name>-RECORD is not of the same type as the one previously retrieved.
4	Record LOCKed.
5	LOCK limit exceeded. The maximum number of records that your program may LOCK is an operating-system parameter. See the "File Status Codes" section of ~Interactive COBOL Programmer's Reference~.
7	Deletion of a parent record and its subtree of dependent records was not completed. This condition occurs when a record in the subtree is LOCKed when the DELETE-<record-name> procedure is executed. (FMS code tries ten times to delete a LOCKed record, hoping it will be UNLOCKed before the final attempt.)
9	End of file.

Included on the FMS release medium is a demonstration program named DEM$FMS. This program was created to process the purchases/payments database used in this manual to illustrate FMS database structures.

Also included on the release medium is a small database created and processed by the program, and the FMS work files that store the database specifications (Data Element Dictionary, Record type descriptions, etc.). Thus, you can both use the program itself and use the FMS to retrace the steps that created the database structure.

To load DEM$FMS from the release tape or diskette, consult the FMS Release Notice.

Compiling DEM$FMS

The DEM$FMS program has already been compiled and is ready to execute. You may wish to recompile the program, however, in order to obtain a printed listing. In Chapter 4, the examples of FMS procedures include line/page annotations. These refer to the compilation listing of DEM$FMS produced by the following CLI commands:

```
COBOL/C/L/P/X DEM$FMS.CO
PRINT DEM$FMS.LS
```

The six-character line numbers that begin all source lines have been edited so that you can easily distinguish programmer-written code from the various FMS-written sections. The edited line numbers are as follows:

line number	meaning
000000	programmer-written line
CCCCCC	programmer-written line that calls an FMS procedure
PCPCPC	line of an FMS "program copyfile"
DCDCDC	line of an FMS "database copyfile"

Running DEM$FMS

To run the demonstration program, choose the "RUN PROGRAM" function of the LOGON menu (or the comparable MASTER menu choice). Figure A.01 illustrates the general flow of control in DEM$FMS operation. Following are some general instructions for using the program:

Flow of Control

Use <CR> to terminate all entries. At any time the <ESC> key cancels the current data-entry or menu-choice operation and "pops" you up to the next higher level.

Key Specification

* To identify a CUSTOMER-RECORD, you enter a level-0 key value at the "Customer Id" prompt.

* To identify a PURCHASE-RECORD, you enter a level-0 key value at the "Customer Id" prompt, and a level-1 key value at the "Purchase Order Number" prompt.

* To identify a PAYMENT-RECORD, you enter a level-0 key value at the "Customer Id" prompt, and a level-1 key value at the "Invoice Number" prompt.

Function Verification

The ADD, CHANGE, and DELETE routines all require a "Y" response to a line-24 "Verify ?" prompt before proceding to perform the appropriate FMS database access procedure.

The DISPLAY Routine

"Display customer records" displays all customer records in the database. "Display purchase records" displays the purchase records for a particular customer -- you identify the customer by entering the "Customer Id". "Display payment records" works similarly.

The FMS generates Interactive COBOL source code in the two routines "Generate Data Base Copyfiles" and "Generate Program Copyfiles".

* DATABASE COPYFILES contain source code that you can include in your program(s) with appropriate COPY statements. A file of this type contains code that refers to a particular ~database structure~, to a particular ~record type~, or to a particular ~data file~ used to store database records.

* PROGRAM COPYFILES themselves contain the COPY statements mentioned above that name the database copyfiles. Hence, they are not true COPY files -- the only way to incorporate the contents of a program copyfile into a program is to use a CRT/EDIT or IC/EDIT "insert file contents" command.

The tables below list the contents of all source code files created by the FMS.

Database Copyfiles

Files Created for Each Database Structure

<database-name>.SD	Documentation file: summarizes all other database copyfiles
<database-name>.SS	SELECT statement
<database-name>.SF	FD statement(s) to define the "generic" RECORD AREA for the data file(s) that store the database.
<database-name>.SW	Working-Storage copy of the FD generic RECORD AREA. This area is used by FMS procedures and is transparent to the programmer.
<database-name>.SP	Procedures that apply to the entire database: LOCK-NEXT-<database-name>-RECORD UNLOCK-<database-name>-RECORDS <database-name>-SEQUENTIAL-LOCK-CHECK <database-name>-RECORD-TYPE

Database Copyfiles (cont.)

Files Created for Each Record Type

<record-name>.$W	Working-Storage work area: <record-name>-RECORD
<record-name>.$V	Clearing record WSC-<record-name>-RECORD
<record-name>.$I	Procedures required for a read-only record type: RETRIEVE-<record-name> GET-NEXT-<record-name>
<record-name>.$O	Procedures required for a read-write record type: RETRIEVE-<record-name> GET-NEXT-<record-name> INSERT-<record-name> REPLACE-<record-name>
<record-name>.$X	Procedure to delete a record: DELETE-<record-name>

Files Created for Each Data File

<file-name>.$I	Procedures required for a read-only database: READ-<database-name> READ-<database-name>-ALTERNATE-KEY RETRIEVE-NEXT-<database-name>-LEVEL-1 RETRIEVE-<database-name>-NEXT-RECORD START-<database-name> SET-<database-name>-ALT-KEY-1-ON SET-<database-name>-ALT-KEY-1-OFF
<file-name>.$O	Procedures required for a read-write database: READ-<database-name> READ-<database-name>-ALTERNATE-KEY RETRIEVE-NEXT-<database-name>-LEVEL-1 RETRIEVE-<database-name>-NEXT-RECORD START-<database-name> SET-<database-name>-ALT-KEY-1-ON SET-<database-name>-ALT-KEY-1-OFF WRITE-<database-name> REWRITE-<database-name>
<file-name>.$X	Procedure to delete a record: DELETE-<database-name>

Program Copyfiles

<program-name>.$S	* Copies of the contents of all files of named <file-name>.$S.
<program-name>.$W	* General purpose Working-Storage data-items * COPY statements to include database copyfiles named <file-name>.$W * COPY statements to include database copyfiles named <record-name>.$W
<program-name>.$P	* COPY statements to include database copyfiles named <file-name>.$P * COPY statements to include database copyfiles named <record-name>.$P
<program-name>.$F	* COPY statements to include database copyfiles named <file-name>.$F