Using the VAX Information Architecture

Order No. AA-GR93A-TE
Including: AD-GR93A-T1

January 1986

This volume introduces the VAX Information Architecture family of software products and gives a step-by-step introduction to developing applications with the products. It also includes a documentation directory, master glossary, and master index for documentation of the VAX Information Architecture family of products.

OPERATING SYSTEMS: **VMS**

MicroVMS

digital equipment corporation, maynard, massachusetts

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: P&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
P&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

VAX Information Architecture Documentation Directory, Master Glossary, and Master Index

---

January 1986

This manual describes the documentation available for the VAX Information Architecture family of software products. It also includes a master glossary and a master index to the documentation sets.

OPERATING SYSTEMS: **VMS**

MicroVMS

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

ACMS	DECUS	UNIBUS
CDD	MicroVAX	VAX
DATATRIEVE	MicroVMS	VAXcluster
DEC	PDP	VAXinfo
DECgraph	Rdb/ELN	VAX Information Architecture
DECnet	Rdb/VMS	VMS
DECslide	TDMS	VT

digital™

VAX Information Architecture
Summary Description

December 1985

This document describes the components of the VAX Information Architecture.

OPERATING SYSTEM: **VMS**

MicroVMS

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

ACMS	DECUS	UNIBUS
CDD	MicroVAX	VAX
DATATRIEVE	MicroVMS	VAXcluster
DEC	PDP	VAX Information Architecture
DECgraph	Rdb/ELN	VMS
DECnet	Rdb/VMS	VT
DECslide	TDMS	digital™

Contents

3 VAX Rdb/VMS

4 VAX DBMS

Index

Figures

Tables

How to Use This Manual

This manual describes the components of the VAX Information Architecture. You should use it to familiarize yourself with these products.

Intended Audience

This book is intended for the DP professional who wants to become acquainted with the components of the VAX Information Architecture. You do not need expertise with the individual components of the VAX Information Architecture to begin reading this book. However, you should have some familiarity with the VMS operating system, VAX Record Management Services (RMS). If you do not, you can read:

- The *Introduction to VAX/VMS* for general information about the VMS operating system
- The *Guide to VAX/VMS File Applications* for information about VAX RMS file handling
- The *VAX Software Handbook* for an overview of VAX facilities and capabilities

To verify which versions of your operating system are compatible with these versions of the VAX Information Architecture products, check the most recent copy of the following:

- For the VMS operating system -- *VAX/VMS Optional Software Cross Reference Table*, SPD 25.99.xx
- For the MicroVMS operating system -- *MicroVMS Optional Software Cross Reference Table*, SPD 28.99.xx

Structure

There are seven chapters in this book:

Chapter 1	Provides an introduction to the VAX Information Architecture.
Chapter 2	Describes the VAX Common Data Dictionary (CDD).
Chapter 3	Describes the VAX Relational Database Management System (Rdb/VMS).
Chapter 4	Describes the VAX Database Management System (DBMS).
Chapter 5	Describes the VAX Terminal Data Management System (TDMS).
Chapter 6	Describes the VAX Application Control and Management System (ACMS).
Chapter 7	Describes VAX DATATRIEVE.

An Overview of the VAX Information Architecture 1

Businesses today have to manage ever-increasing quantities of information. Controlling inventory, tracking customer credit, filing reports with government regulatory agencies, and analyzing business trends are all examples of managing information. But what exactly does "information management" mean? The next few pages provide an answer.

1.1 Information Management

Requests for information have been increasing steadily in recent years. The trend has led managers to seek solutions outside the data processing department. Instead of submitting all requests to a central group, department heads have begun to hire their own programmers to develop departmental applications.

Decentralizing data processing in this way increases overall efficiency, but at the expense of control. Traditional data processing requires that each application program describe the data and how it is used within the logic of the program. Programs and data, therefore, can be so dependent on one another that a change in one requires a change in the other. Redundancy and inconsistency result when different departments process data independently. Instead of accessing the central files, departmental programmers often duplicate data stored in the central files for their own applications. Subsequent updates to the central files are not included in the local copies, so that files become less and less reliable over time.

To let organizations maintain control over data processed locally by different departments, software products have been developed that keep the definition and management of data separate from application programs. With these information management products, individual departments no longer need to maintain their own data files, nor must data access originate in a central data processing department. Instead, processing can take place locally, while the software protects data against unauthorized access, redundancy, and inconsistency.

Information management makes it possible for users outside the data processing department to get needed information without concern for the details of its physical storage. Office workers and managers can examine data and format it as useful information. Different departmental data processing groups can simultaneously update the central files without interfering with one another. Programmers can update programs without having to redefine the files in which data is stored.

Correctly implemented, information management software can improve the overall efficiency of an organization's data processing. Relieved of the necessity of answering numerous *ad hoc* requests from users, the central data processing department can devote full attention to designing and maintaining structured applications. In other departments, information management tools let users develop their own applications to answer their own information needs.

1.2 Information Management Tools

Many software tools are available for setting up efficient information management systems. These tools perform the following functions:

- Information resource management, providing central storage of data descriptions and record definitions
- Data access, letting you easily retrieve information
- Distributed processing, allowing you to process data stored on other computers remote from your local system
- Reports and graphics generation, providing informative and attractive reports, graphs, and charts
- Terminal management, displaying familiar business forms on the terminal screen to make it easy to manipulate the data in your files
- Data and database management, controlling data shared by many users
- Application management, controlling large, complicated applications

1.2.1 Information Resource Management

Data in files is described by record definitions. Traditionally, these definitions have been included in the programs that process the data. The COBOL data division, for example, contains definitions for all of the data used in a COBOL program. Information resource management helps avoid a proliferation of files containing the same data defined differently. This approach makes data descriptions independent of program logic.

The principal tool of information resource management is the **data dictionary**. Data dictionaries define and describe all of the data items used by an organization. Instead of creating new files and record definitions as they perceive a need, programmers can use the data dictionary and the information resources that already exist.

Data dictionaries can be active or passive. Passive dictionaries simply store descriptions of data and generate listings of data definitions and available information resources. Active data dictionaries let programs extract data definitions as program source code. With an active data dictionary, you can create new applications, or modify old ones, without redefining data. Instead, you can include the dictionary definition automatically in your application regardless of language. Use of an active data dictionary increases efficiency and maintainability by reducing the number of program-specific definitions.

1.2.2 Data Access

To make it easy to retrieve data for processing, special query languages let you use everyday English words to perform tasks that used to require application programs. Query language capabilities include:

- Searching files for information based on criteria you specify
- Sorting data
- Adding data
- Modifying data
- Deleting data
- Protecting data

For example, a query language lets you use a simple command to find the names of all your employees who earn between $25,000 and $30,000 a year:

```
PRINT EMPLOYEES WITH SALARY BETWEEN 25000 AND 30000
```

The query language finds all employees matching the criteria and displays information about them on your terminal screen.

1.2.3 Distributed Processing

Distributed processing gives you the ability to access data on remote computers as easily as you access data stored on your local computer. With distributed processing, you can decentralize your data files without introducing redundancy or relinquishing control.

In a distributed processing environment, the data your department uses most frequently is stored locally. When a user on another computer needs to access your data, distributed processing software handles the physical data retrieval. From the user's point of view, there is no difference between local and remote processing. Distributed processing helps prevent data redundancy because no new copies of the original data files are made.

1.2.4 Report and Graphics Generation

Report writers make it easy to retrieve data from central files or databases, to arrange and manipulate that data, and to produce informative and attractive reports.

For example, a simple summary report might involve printing the name and monthly revenues of each branch of a department store chain followed by the total monthly revenue. Producing this report with a traditional programming language requires the following steps:

- Create a variable TOTAL_REVENUE and set its value to 0.
- For each branch, add the monthly revenue to TOTAL_REVENUE and then print the values of NAME and REVENUE in the report.
- After the last branch has been processed, print the value of TOTAL_REVENUE in the report.

With a report writer, you do not need to calculate the total explicitly. Instead, simple commands specifying what you want, not how to produce what you want, are all that is required:

```
PRINT NAME, REVENUE
AT BOTTOM PRINT TOTAL_REVENUE
```

In the sample report in Figure 1-1, the software displays the names of salespeople sorted into groups based on length of employment and performance against sales quotas. The report writer performs all of the sorts and calculations automatically.

Most report writers let you store report formats for future use, so that once you have defined a format, you can use it later to produce reports automatically. Whether you need a report that is used only once or a report that the government requires you to file each month, a report writer lets you produce the report quickly and easily.

Graphics generators are similar to report writers, but, instead of producing reports, they present the data stored in your files as line and scatter graphs, bar charts, and pie charts. To allow you to create graphs without having to write programs, graphics generators usually offer a simple command syntax or a menu interface for graphic design.

SALES COMMISSION REPORT 2-Jul-1985
Page 1

RATING	COMM PCT	SALES NAME	MONTHS EMP	AMOUNT	COMMISSION
BELOW QUOTA	5%				
		ANNE DINNAN	3	$2,389.90	$119.50
		RICK LANGHART	4	$4,999.99	$250.00
		LYDIA BARNET	1	$2,598.79	$129.94
		JOSEPH FREDERICK	4	$5,000.00	$250.00
		NUMBER: 4	TOTAL SALES:	$14,988.68	$749.53
BELOW QUOTA	7%				
		WILLIAM SULLIVAN	9	$8,672.99	$607.11
		LINDA REINE	7	$8,532.22	$597.26
		HENRY MAILER	7	$9,999.99	$700.00
		NUMBER: 3	TOTAL SALES:	$27,205.20	$1,904.36
ABOVE QUOTA	10%				
		NANCY ROTHBLATT	2	$6,325.88	$632.59
		WAYNE SMITH	5	$9,853.52	$985.35
		SEYMOUR KIMMELMAN	5	$7,325.67	$732.57
		NUMBER: 3	TOTAL SALES:	$23,505.07	$2,350.51
ABOVE QUOTA	12%				
		DAN DERRICK	8	$11,456.87	$1,374.82
		JAMES STORER	14	$25,876.02	$3,105.12
		SANDY LEVINE	8	$10,000.01	$1,200.00
		DENNIS MCADOO	11	$12,345.62	$1,481.47
		NUMBER: 4	TOTAL SALES:	$59,678.52	$7,161.42
**********					**
		SALES FORCE: 14	TOTAL SALES:	$125,377.47	$12,165.73

Figure 1-1: A Sample Report Produced by a Report Writer

Graphics are a dramatic way to change data into information; large quantities of information can be grasped at once, and trends quickly become apparent. Consider, for example, the difference between reading columns of figures and seeing a graph of those figures over time. Graphics programs can quickly produce

sophisticated color displays of your data. For example, Figure 1-2 is a pie chart that displays the percentage of employees in each department within a company. The chart was produced with the command:

```
PLOT PIE USING DEPARTMENT OF PERSONNEL
```

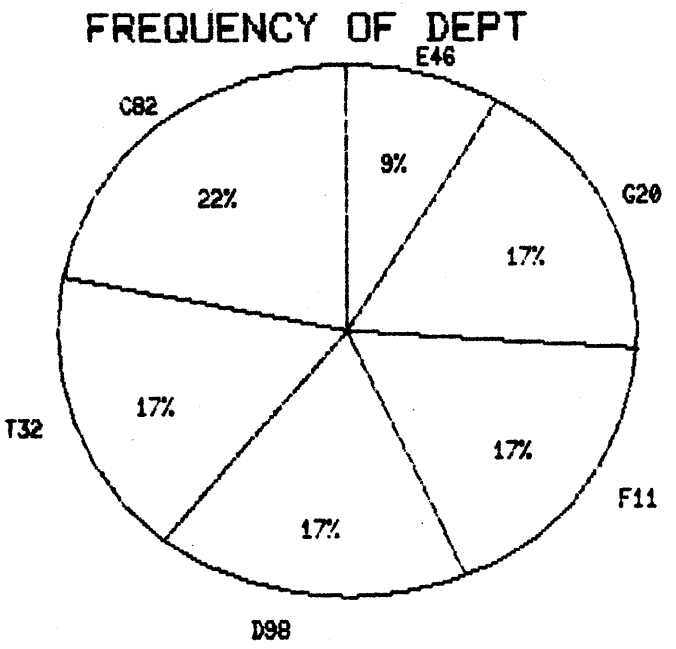


Figure 1-2: A Sample Pie Chart Produced by a Graphics Generator

As with report writers, you can experiment with the graphs and easily make changes until you have the graph you want.

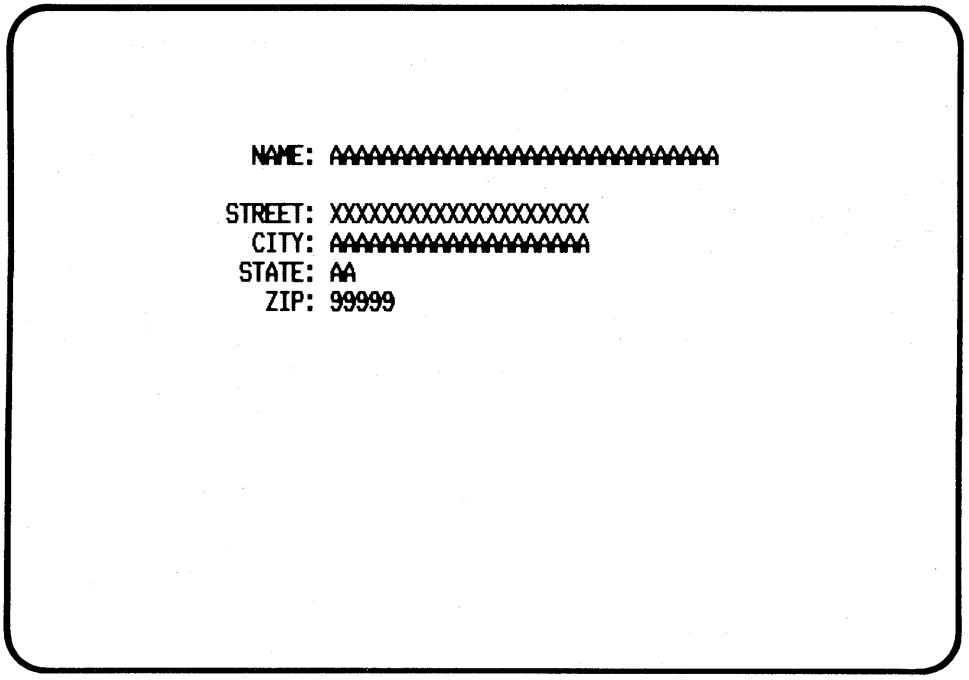
See Chapter 7 for more information about producing reports and graphics with VAX DATATRIEVE software.

1.2.5 Terminal Management

In business, most data is gathered and stored on forms. Displaying business forms on a terminal screen provides a familiar and easy method for entering and retrieving data.

Many forms processors check values as the data is entered and accept a value only if it is of a specified type or within a specified range. For example, you can direct the form to accept a value for an employee code only if that value corresponds to one of the values listed with the form definition. Control of this kind leads immediately to fewer data entry errors.

Figure 1-3 is an example of a form. An employee at a terminal enters data into the fields defined on the form or reads the information displayed there by the forms processor.



ZK-00060-00

Figure 1-3: A Sample Form

The fields of this form are filled with characters ("A", "X", and "9") that determine the field length and the types of characters allowed in that field. Thus, "A" specifies alphabetic characters, "9" specifies numeric characters, and "X" specifies alphanumeric characters.

1.2.6 Database Management Systems

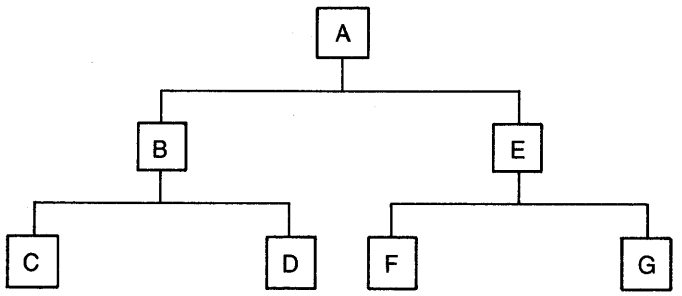
During the 1970s, sophisticated database management systems (DBMS) emerged to provide greater control over data than that available with conventional file structures.

In general terms, a database is simply stored data, but the term has a more specialized meaning in the context of database management systems. Like traditional files, DBMS files contain data and record definitions, but DBMS files also contain representations of the relationships among the data items and records. Instead of relying on traditional file access methods, DBMS software controls access to data and data definitions. There are three basic database structures:

- Hierarchical
- Network, also called the CODASYL model because the Conference on Data Systems Languages has been active in developing network database specifications
- Relational

A hierarchical database organizes the relationships between record types as a tree structure. Related records are stored on the same branch of the hierarchy to facilitate efficient data retrieval. A disadvantage of the hierarchical structure is the lack of flexibility in navigating through the database: once you choose one of the branches, there is no way to get to the records on the other side of the branch without moving back up the tree to the junction of the required branch. At that point you can begin working down the other side of the tree.

In Figure 1-4, the hierarchical relationships are clear.



ZK-00061-00

Figure 1-4: The Hierarchical Database Model

Records C and D are clearly related to Record B, which is, in turn, related to Record A. If you want to relate Record C to Record E, however, the hierarchical organization of the database requires you explicitly to link C to B, B to A, and A to E when you access these records in a program.

With the network or **CODASYL** model, any record can be related to any other record without the restrictions inherent in the hierarchical structure. Because records can participate in relationships, called sets, that are not limited to records hierarchically above and below, the network model provides flexibility in matching database structures to your data processing needs.

In Figure 1-5, the EMPLOYEE record participates in three set relationships.

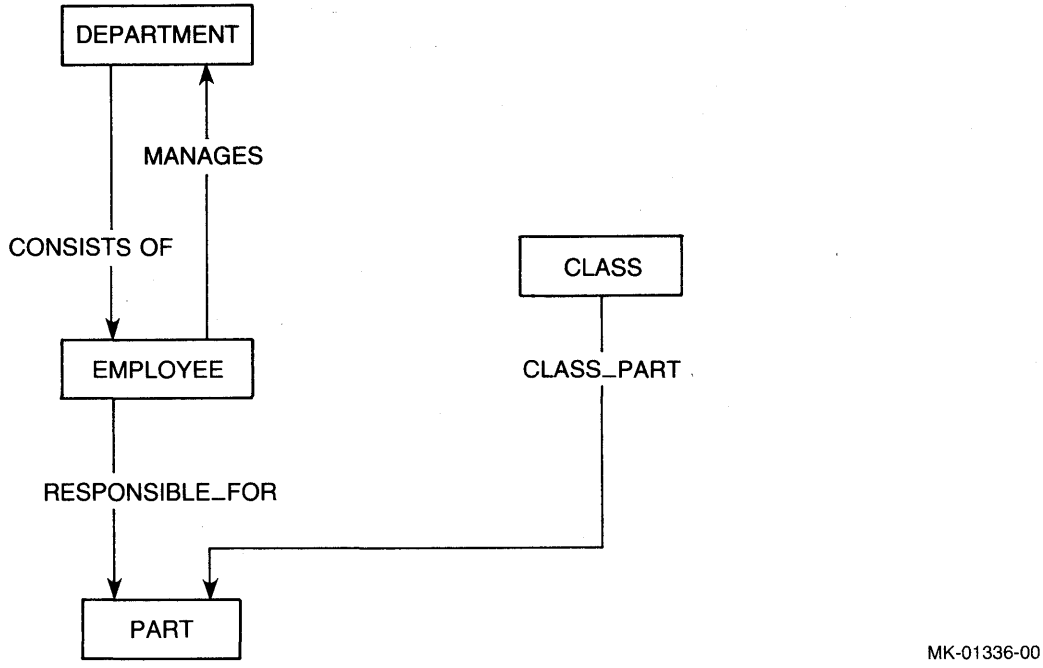


Figure 1-5: The Network (CODASYL) Database Model

Departments both consist of employees and are managed by them. In addition, employees are responsible for maintaining parts. By adding record types and sets in this way, you can use a network database to reflect the data relationships in your organization. The advantage of predefining relationships in network databases, especially databases containing large numbers of records, is processing efficiency.

The relational database model provides more flexibility than either the hierarchical or the network model because relationships do not exist as predefined structures. Instead, data is stored in tables, and relationships between two or more

records are established by matching the values of key fields common to those records, as shown in Figure 1-6.

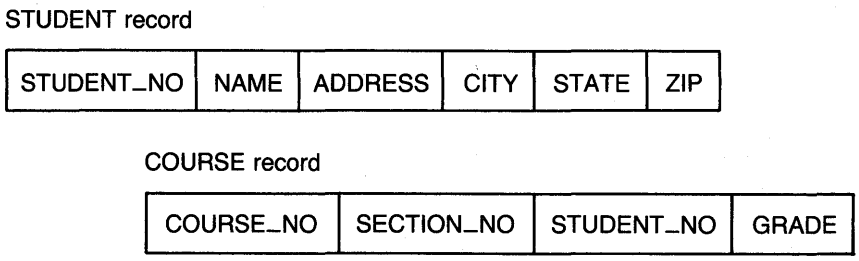


Figure 1-6: The Relational Database Model

In Figure 1-6, no relationships between students and classes are defined. Because the student number is common to both records, however, it is easy to associate a class number and grade with the name and address of the student who took that course and earned that grade.

In summary, implementation of a database management system can provide several benefits:

- Reduction in redundancy

  Instead of storing several copies of the same data in each of several files, a DBMS stores data and data definitions in central files and controls the physical storage.

- Views

  Individual users see only those portions of the database they need to do their work. These subsets of the database are called logical views.

- Security

  DBMS software enforces security. If some of your data is sensitive, you can ensure that only authorized personnel can read or change it.

- Shared access

  Because the database software controls access to the data, it is possible to control shared access to files. This means that many of your employees can update the database simultaneously without introducing errors. Database management software is programmed to resolve any conflicts that might arise.

- Recovery from failure

  As employees process database data, their transactions are recorded in a journal file. Therefore, the database can be restored to accuracy if hardware or system failures corrupt or destroy a day's database activity.

Database management systems provide these benefits because they perform many of the data handling and file control functions that must be performed by individual programs in a conventional file management system. Conversion to a database from a conventional file system, however, can be expensive at first, in part because trained technical personnel are sometimes needed to design and implement database applications.

1.2.7 Comparison of Relational and CODASYL Databases

You use VAX DBMS for applications in which:

- Databases are large. VAX DBMS was designed to handle databases up to several gigabytes.
- The relationships between different parts of the database range from normal to very complex. VAX DBMS is appropriate for databases with 30 or more record types and set relationships.
- Records and their various relationships are clearly understood during the design phase.
- Relationships are relatively stable.

VAX DBMS requires the expertise of database designers, programmers, and administrators. The users of VAX DBMS are usually experienced database designers and programmers.

VAX DBMS is most efficient when the benefits of performance tuning and stable relationships offset the additional time and effort spent planning and implementing database applications.

You use VAX Rdb/VMS for applications in which:

- The structure of the database is expected to change significantly over time. An application requiring frequent prototyping, for example, benefits from the relational structure of VAX Rdb/VMS.
- Application programmers, rather than experienced database designers and administrators, create and use the database.
- Remote database access and distributed database workloads are desirable.

VAX Rdb/VMS is most efficient when changes in the database are normal and desirable. Rdb/VMS makes the restructuring of relationships easy and quick.

Note that the relationship between DBMS and Rdb/VMS is not necessarily either-or. A single application system might have some parts that are well-suited for a relational database and other parts that are well-suited for a CODASYL database.

1.2.8 Application Management

As your information management system grows and becomes accessible to more and more employees, you need more control and more efficient processing of common data. Application management systems answer this need. Typically, application management systems let employees with little or no computer experience perform standardized data processing tasks by making selections from a menu displayed on a terminal screen.

Application management systems give you broad control over which menus each of your employees can see and use and over the tasks each employee can perform. These systems also provide facilities for logging the work users have done. Such data is necessary both for application security and for tuning application programs. With application management systems, you gain the benefits of efficient data processing while minimizing the risk of granting broad access to your company data.

1.3 Planning an Information Management System

Each of the tools previously described provides benefits, but no business information problem has only one solution. Different tools, and groups of tools, solve different problems. For example, a query language and report writer might provide all of the information management needed by a company of 50 employees. On the other hand, a company that manufactures and distributes more than 1000 products should certainly investigate the benefits of a database management system. To make an intelligent choice, you must evaluate the information management products available in light of your particular business needs.

The following sections introduce DIGITAL's family of information management products, the VAX Information Architecture. In addition to learning about the products, you will see how they work together in different combinations to answer different needs. Later chapters discuss individual components of the VAX Information Architecture in greater detail.

1.4 What is the VAX Information Architecture?

The following sections describe the VAX information management products and the information management problems they solve. The VAX Information

Architecture includes:

- VAX Common Data Dictionary, DIGITAL's central storage facility for data definitions used by VAX Information Architecture products and a growing number of VAX languages
- VAX Rdb/VMS, DIGITAL's relational database management system
- VAX DBMS, DIGITAL's CODASYL-compliant database management system
- VAX TDMS, DIGITAL's terminal management package that displays forms and manages data using definitions stored in the CDD
- VAX ACMS, DIGITAL's software for application management and development
- VAX DATATRIEVE, DIGITAL's query and report writing language

VAX Information Architecture products work with each other, and with VAX native-mode languages conforming to the VAX calling standard, to provide flexible solutions to your information management problems.

1.4.1 VAX Common Data Dictionary

The VAX Common Data Dictionary (CDD) provides central storage for all of the data descriptions used and shared by VAX Information Architecture products and by most VAX high-level languages. This sharing of data descriptions provides several benefits:

- Modifications to data definitions can be made easily because all definitions are centrally located. For example, if the United States Postal Service changes ZIP codes from five digits to nine, data files, forms, and data definitions will have to be restructured. Once the files and forms are changed, the CDD user will need only to change those record definitions that include the ZIP code field and to recompile the programs that use them. Programmers using a conventional file system without shared data definitions will have to modify every program that contains a reference to ZIP codes.
- VAX Information Architecture products can use the same data and the same files because the definitions describing record structures can be shared. For example, you can use a CDD record definition in VAX COBOL to read and process a file created by VAX DATATRIEVE. You do not have to write special programs to allow the different products to work together or store redundant copies of data files, each suited to a specific product.
- When a user or a program accesses a definition in the dictionary, you have the option of keeping a record of that access in a history list. With the

CDD's history list feature, you can keep a record of dictionary usage. For example, history lists could provide the names of all the programs that included a particular record definition at compile time, and this information would help you assess the impact of changing that definition.

- The CDD has a security mechanism that allows you to protect the definitions in your dictionary against unauthorized access or modification.

The VAX Common Data Dictionary has a hierarchical directory structure consisting of one or more physical files. The CDD includes three utilities that let you organize your dictionary and store data definitions in it:

- The Dictionary Management Utility (DMU) provides a set of commands that let you create, back up, copy, and protect your dictionary hierarchy.
- The CDD Data Definition Language Utility (CDDL) lets you store shareable record descriptions.
- The CDD Verify/Fix Utility (CDDV) verifies dictionary files and repairs some file corruptions resulting from hardware failures or I/O errors. It can also rearrange dictionary files to reduce their size and improve performance.

The VAX Common Data Dictionary can be simultaneously accessed and updated by many concurrent users. The CDD uses the Lock Manager facility of the VMS operating system to guarantee that users do not interfere with one another.

With the CDD and the VAX Information Architecture tools described in the following sections, you can choose the products you need to solve your business problems.

See Chapter 2 for more information about the CDD.

1.4.2 VAX Rdb/VMS

VAX Rdb/VMS is a relational database management system for VAX computers using the VMS operating system. Rdb/VMS gives you the advantages of a full-featured database management system, including data security and integrity and optimized access. Because Rdb/VMS uses the relational model of data storage, Rdb/VMS is flexible and easy to use.

VAX Rdb/VMS provides the following features:

- Rdb/VMS makes it easy for experienced programmers to design and restructure databases. In most cases, you do not need a professional database administrator to create and maintain a database.

- Many users can retrieve information from the database and update it simultaneously.
- Before- and after-image journaling ensures that the accuracy and reliability of the database is maintained in the event of user errors and hardware or software failures.
- The DIGITAL Standard Relational Interface (DSRI) allows programs written for VAX Rdb/VMS to run on VAX Rdb/ELN software, a relational database management system using the VAX/ELN operating system. Similarly, programs written for VAX Rdb/ELN software run without modification on VAX Rdb/VMS.
- Programs written for VAX Rdb/VMS can access information in local databases and in remote Rdb/VMS or Rdb/ELN databases.
- Rdb/VMS can store its definitions in the VAX Common Data Dictionary so that you can use VAX DATATRIEVE to query the database and produce reports and graphs.
- Security mechanisms let you control access to Rdb/VMS elements and data.
- Precompilers for VAX BASIC, VAX COBOL, VAX FORTRAN, and VAX PASCAL let you include VAX Rdb/VMS statements in programs written in any of these languages. For languages unsupported by precompilers, Rdb/VMS provides an interpretive call interface.
- An interactive utility, the Relational Database Operator (RDO), lets you maintain the database, create and modify definitions of database elements, and store and manipulate data. When you type RDO statements, Rdb/VMS executes those statements immediately.

See Chapter 3 for more information about VAX Rdb/VMS.

1.4.3 VAX DBMS

VAX DBMS is a sophisticated CODASYL-compliant database management system that lets many users simultaneously to retrieve and update data stored in the same database files. Typically, VAX DBMS applications involve:

- High-volume retrieval and update
- Multi-user access to the same data
- Relatively stable applications using the data

Database management software provides efficient use of a computer's processing abilities, but it requires careful planning and implementation.

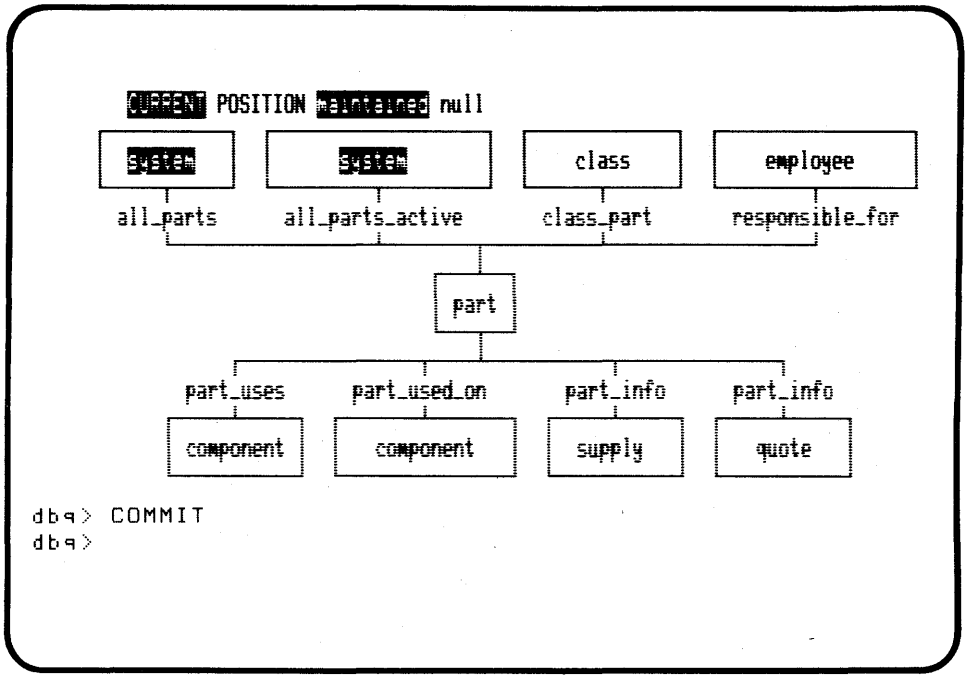
VAX DBMS provides the following features:

- VAX DBMS creates CDD definitions for the logical definition of the database, for application program views of this logical structure, and for the physical structure of the database on mass storage media. These data definitions are, therefore, kept separate from application programs.
- Security provisions let you control access to VAX DBMS data by defining the access privileges for applications using VAX DBMS databases.
- Many users can retrieve and update the database simultaneously.
- Before- and after-image journaling ensure that the accuracy and reliability of the database is maintained in the event of user errors and hardware or software failures.
- A high-level call interface makes the data stored in VAX DBMS databases available to VAX DATATRIEVE.
- The interactive database query utility (DBQ) provides data manipulation capabilities (CONNECT, DISCONNECT, ERASE, FETCH, FIND, GET, MODIFY, RECONNECT, and STORE) and simultaneous video display of application views of the database.
- Programmers use the VAX DBMS data manipulation language (DML) to access a database. DML is understood by the VAX COBOL language and is available to FORTRAN programmers through FORTRAN/DML, a VAX DBMS preprocessor to the VAX FORTRAN compiler. VAX DBMS also has a precompiler that lets you insert DML statements into programs written in the following languages:

  ```
  VAX BASIC
  VAX BLISS
  VAX C
  VAX DIBOL
  VAX PASCAL
  VAX PL/I
  ```

- The database operator facility (DBO) lets you manage your databases through a command language that is easy to learn and simple to use.

Figure 1-7 shows a subschema as displayed by the DBQ utility.



ZK-00063-00

Figure 1-7: VAX DBMS Subschema Display

You can convert most existing applications based on a CODASYL DBMS to VAX DBMS with relatively little effort. The basic design of the application usually need not change. You need only:

- Convert the schema, subschemas, and storage schema
- Convert the application programs
- Move the existing data with the VAX DBMS Load Utility

Conversion from nondatabase applications, however, involves a great deal of effort in database and application design. This is a major part of the cost of acquiring a DBMS package.

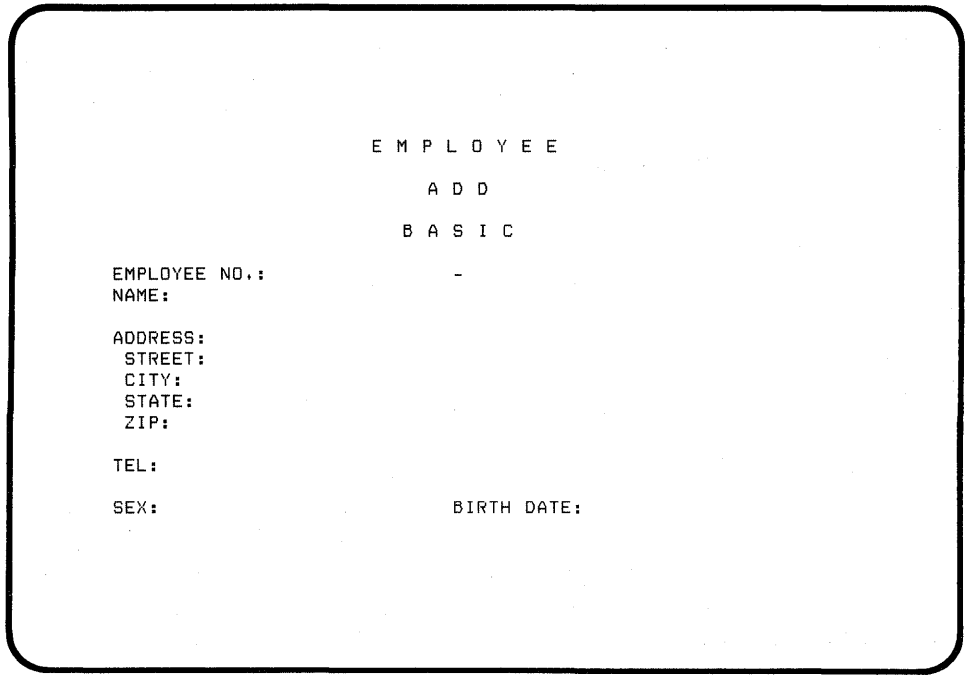
See Chapter 4 for more information about VAX DBMS.

1.4.4 VAX TDMS

VAX TDMS expands the traditional concept of forms management to include control of all input and output. With VAX TDMS, a special data structure, called a **request**, associates a form definition with a record definition. Within the request, you can include instructions (for input and output, for checking value ranges, and for testing whether possible conditions are true) that would otherwise have to be included in applications programs. Request, form, and record definitions are all stored in the CDD.

You create VAX TDMS forms by designing them directly on your terminal screen. You do not need complex charts as an intermediate step or a special forms design language. With VAX TDMS, you can modify forms at any time without having to make complicated changes to your program code, and you can change your programs without having to modify your forms.

Typical VAX TDMS applications range from database inquiry and update to the periodic display of the status of an industrial process. TDMS forms can be used to help clerical personnel easily enter data at the terminal. You can also use TDMS forms to provide menus for data entry or for the selection of different program options in an application. Figure 1-8 is an example of a form produced by VAX TDMS.



ZK-00064-00

Figure 1-8: ADD_EMPLOYEE_FORM: A Sample VAX TDMS Form

The following sample request is part of a personnel administration application. The request links fields from the form in Figure 1-8 to fields in a record definition named PERS_RECORD:

```
CREATE REQUEST ADD_EMPLOYEE_REQUEST
   FORM IS ADD_EMPLOYEE_FORM;
   RECORD IS PERS_RECORD;
   USE FORM ADD_EMPLOYEE_FORM;
   INPUT NUMBER  TO PERS_NUMBER,
         FIRST   TO PERS_FIRST,
         INITIAL TO PERS_INITIAL,
         LAST    TO PERS_LAST,
         STREET  TO PERS_STREET,
         CITY    TO PERS_CITY,
         STATE   TO PERS_STATE,
         ZIP     TO PERS_ZIP_CODE,
         PHONE   TO PERS_TELEPHONE,
         SEX     TO PERS_SEX,
         BIRTH   TO PERS_BIRTHDATE;
END DEFINITION;
```

Managing information with VAX TDMS provides three major advantages:

- Lower programming costs

  Creating and storing definitions outside of application programs significantly reduces programming and maintenance costs. Because form, record, and request definitions are not written as part of the program, it is often possible to revise your application without changing the application program.

- Data independence

  With VAX TDMS, the application program is independent of the data input/output process. The primary functions of the program are to call and execute requests, provide access to the database that the application uses, and handle errors so that no data becomes corrupted. The applications programmer does not need to be concerned with connecting the data to the forms or the records, because this is done entirely by the request. In many applications, VAX TDMS can reduce the number of programming statements and errors in the application program.

  As a result, you can view the program in a VAX TDMS application as a procedure that executes a series of requests (or routines) and transfers data to and from a database. The requests and form definitions are independent of the program, so you can change them without significant programming costs.

- Device independence

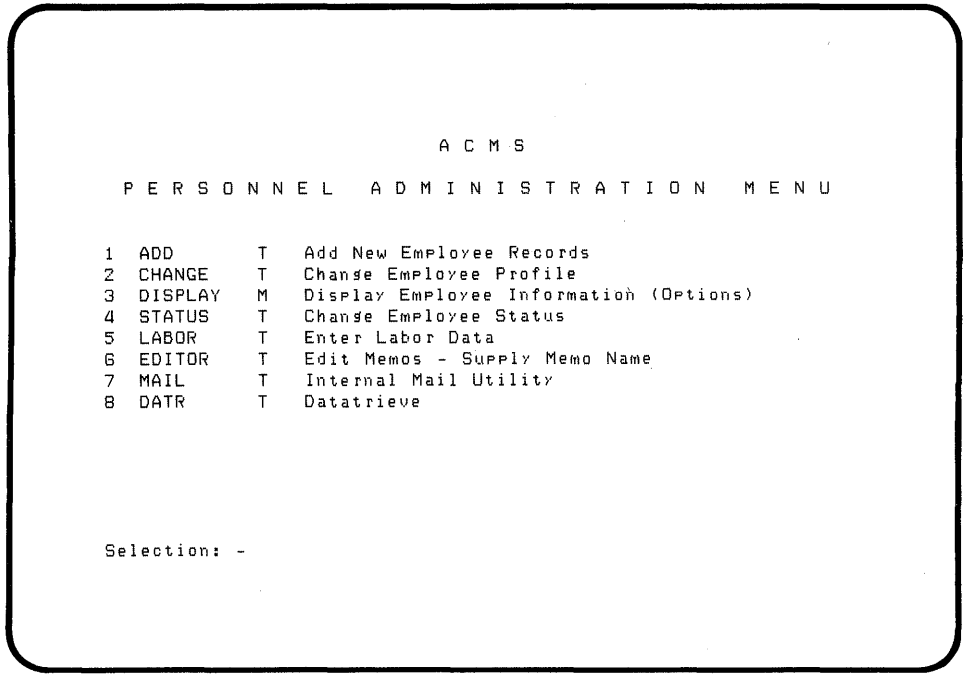
  With VAX TDMS, you do not have to include information about particular video terminal types in application programs. Terminal manipulation (such as cursor control, scrolling, and video highlighting) is defined by the form and the request and is wholly independent of the application program.

See Chapter 5 for more information about VAX TDMS.

1.4.5 VAX ACMS

VAX ACMS is an information management tool that lets you manage complex, multi-user application systems. The typical VAX ACMS application involves simultaneous access to a common database by many users with little or no computer experience. Applications well suited to VAX ACMS include hotel reservation systems, personnel administration systems, and funds transfer systems.

With VAX ACMS, you can create and modify application menus that make it easy for users to select tasks. Figure 1-9 shows a typical ACMS menu.



ZK-00065-00

Figure 1-9: VAX ACMS Menu

Although ACMS supplies a default menu form, you can also use VAX TDMS to design your own menu format and have ACMS use this format for the menus it displays.

With VAX ACMS, you can also:

- Control which users can run which tasks in an application
- Keep track of the volume of tasks run and who runs them
- Keep records of the operations of the system and the resources used by an application
- Add new tasks to an application or new users to a task
- Distribute an application's tasks across a DECnet computer network

You can use VAX ACMS to control applications developed with any of the VAX languages or VAX Information Architecture tools.

For example, with VAX ACMS you can:

- Use VAX TDMS to exchange data between a terminal and predefined workspaces
- Define control fields whose values can trigger error-handling routines
- Define processing steps to specify how data is manipulated
- Ready VAX DBMS subschemas to take advantage of journaling and recovery
- Call subprograms written in VAX native-mode languages to retrieve, store, or modify data in files or in VAX DBMS databases

Traditional programming requires that a task, such as adding a record to the database, be coded in application programs. If the nature or order of the task changes, programs must be completely rewritten. VAX ACMS provides straightforward syntax that lets you define an individual task separately from application programs and to store this definition in the CDD. VAX ACMS tasks call:

- VAX TDMS requests that handle terminal I/O
- Subprograms that control data transfer to VAX RMS files or VAX DBMS databases

VAX ACMS applications are easier to create, easier to understand, and easier to change than standard application programs.

In cases where you can break your tasks down into well-defined sequences of steps, VAX ACMS reduces the quantity of system resources, including memory, used by the task. This savings in memory allows a system running VAX ACMS applications to support more terminals than would be possible if the system were running traditional programs or VAX DATATRIEVE procedures. VAX ACMS lets you create, control, and monitor complex multi-user applications.

1.4.6 VAX DATATRIEVE

VAX DATATRIEVE is a powerful query and application development language, but it has many additional capabilities. With DATATRIEVE, you can:

- Store, update, and retrieve data interactively or with a program
- Generate attractive reports and graphs from data stored in VAX RMS files or in VAX DBMS or VAX Rdb/VMS databases
- Retrieve data from other computers in a computer network as easily as you can from your own computer
- Combine data from two or more files in defined user views or by using the CROSS clause as part of a query or report
- Prototype and test new applications
- Store often-used sequences of statements in DATATRIEVE procedures

The central concept in DATATRIEVE data definition is the domain, which associates the data in files with the appropriate record definitions. Three DATATRIEVE commands--DEFINE DOMAIN, DEFINE RECORD, and DEFINE FILE--create the CDD definitions you need to set up a working DATATRIEVE environment. With the DATATRIEVE STORE statement, you can then insert data into the files you have defined.

With VAX DATATRIEVE, English-like syntax makes data retrieval and *ad hoc* queries easy to learn and easy to use. VAX DATATRIEVE's record selection expressions (RSEs) select the records you want from a domain. Sample RSEs include:

```
EMPLOYEES WITH SALARY GREATER THAN 20000

ACCOUNTS WITH UNPAID_BALANCE GREATER THAN 600

DONORS WITH BLOODTYPE EQUAL O_NEG
```

Compound RSEs are allowed, and you can sort records within an RSE as well. For example:

```
ACCOUNTS WITH UNPAID_BALANCE GREATER THAN 600 AND

DUE_DATE LESS THAN 9/1/82 SORTED BY DUE_DATE
```

Using RSEs with VAX DATATRIEVE statements like PRINT, REPORT, or PLOT produces the information you need in the form in which you need it. For example, the DATATRIEVE statement in Figure 1-10 prints all of the data in a domain named ANNUAL_REPORT.

```
PRINT ALL OF ANNUAL_REPORT SORTED BY DATE
```

DATE	EQUIPMENT SALES	SERVICES	REVENUE	NET INCOME	NET INCOME PER SHARE	RESEARCH	INVENTORIES	EMPLOYE
1971	133.0	13.8	146.8	10.6	0.3	16.7	44.4	7,420
1972	166.3	21.3	187.6	15.3	0.5	20.1	62.1	15,430
1973	229.1	36.4	265.5	23.5	0.7	25.0	102.7	14,226
1974	360.8	61.1	421.9	44.4	1.3	36.6	137.4	14,393
1975	433.2	100.6	533.8	46.0	1.3	48.5	174.8	15,033
1976	586.7	149.6	736.3	73.4	2.0	58.4	218.8	15,442
1977	847.5	211.1	1058.6	108.5	2.8	79.7	375.0	22,738
1978	1128.1	308.5	1436.6	142.2	3.4	115.7	428.1	25,868
1979	1381.8	422.3	1804.1	178.4	4.0	138.3	513.5	28,835
1980	1779.4	588.6	2368.0	249.9	5.4	186.4	819.9	35,117

Figure 1-10: Sample Output of the DATATRIEVE PRINT Command

You can create a revenue report (see Figure 1-11) with the following simple REPORT statement:

```
DTR> REPORT ANNUAL_REPORT SORTED BY DATE
RW> PRINT DATE ("Year"), EQUIPMENT_SALES ("Equipment Sales"),
RW> SERVICES ("Services"),
RW> REVENUE ("Revenue")
RW> END_REPORT
```

```
                    Annual Report                          10-Aug-1985
                      1971-1980                            Page 1

Year      Equipment Sales    Services     Revenue

1971             133.0          13.8       146.8
1972             166.3          21.3       187.6
1973             229.1          36.4       265.5
1974             360.8          61.1       421.9
1975             433.2         100.6       533.8
1976             586.7         149.6       736.3
1977             847.5         211.1      1058.6
1978            1128.1         308.5      1436.6
1979            1381.8         422.3      1804.1
1980            1779.4         588.6      2368.0
```

Figure 1-11: A Sample Report From VAX DATATRIEVE

To display the same information graphically on a VT125 or VT240 terminal, use this PLOT statement:

```
PLOT MULTI_SHADE DATE, REVENUE ("Revenue"),
EQUIPMENT_SALES ("Equipment Sales"), SERVICES ("Services") OF
ANNUAL_REPORT SORTED BY DATE
```

PLOT CROSS_HATCH lets you display shaded areas on a graphics printer, as shown in Figure 1-12.

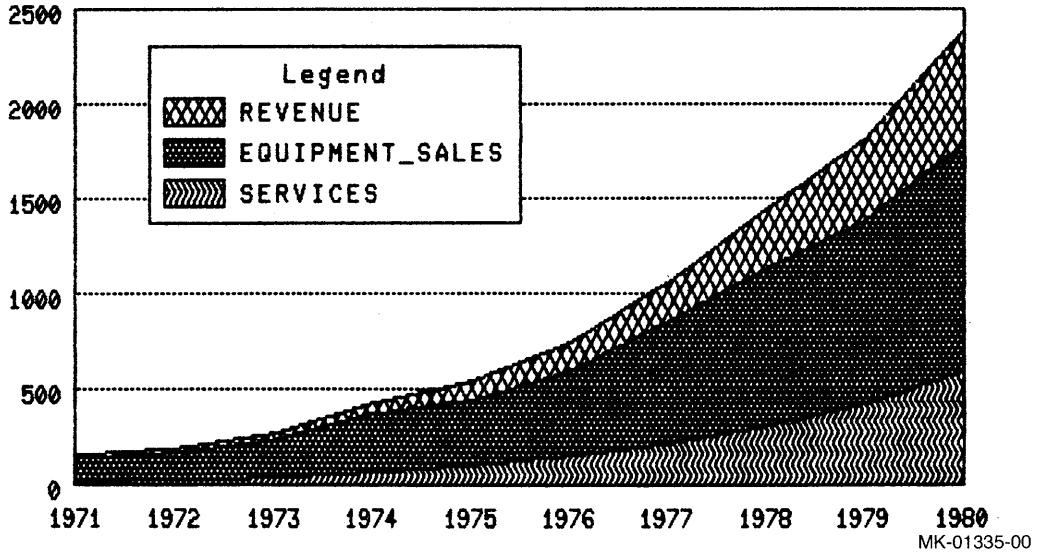


Figure 1-12: A Sample Plot From VAX DATATRIEVE

Three important features make VAX DATATRIEVE easy to use, even if you have little or no programming experience:

- Guide Mode helps you use DATATRIEVE to retrieve or update data by displaying appropriate options at each decision point.
- The Application Design Tool (ADT) simplifies the process of defining domains, record structures, and files. ADT asks you a series of simple questions and uses the responses to build the necessary data definitions.
- The VAX DATATRIEVE Editor lets you modify your data definitions easily. In addition, the Editor lets you correct DATATRIEVE statements and commands that you have entered incorrectly. If the statement fails because of a typing error or because of faulty logic, you do not have to retype the entire statement. Instead, type EDIT, and DATATRIEVE displays the statement ready for editing. After you make the changes and exit from the Editor, DATATRIEVE executes the modified commands and statements.

See Chapter 7 for more information about VAX DATATRIEVE.

VAX CDD 2

2.1 Overview of the VAX Common Data Dictionary

In traditional programming, each program has its own individual data files. Within a program, the programmer defines all the records in the associated data files. This style of programming leads to data redundancy and inconsistency. For example, if ten different programs use the same record, that record definition is likely to appear in all ten programs. Because different programmers may choose to define the same record in different ways, these record definitions soon become inconsistent. Further, if the record definition changes, the source code for all ten programs must also change. If the source code does not change to match the changed definition, the data stored by the separate programs becomes inconsistent.

You can combat the inconsistency resulting from unrestricted use of similar record definitions by using a data dictionary. A data dictionary is a central repository for data definitions. It can:

- Store data definitions
- Keep information about the location of each definition
- Provide a method of access to each definition
- Keep track of what happens to each definition

The VAX Common Data Dictionary (CDD) is a central repository for data descriptions and definitions that can be shared by VAX languages and VAX

Information Architecture products. Using the CDD, a data administrator can:

- Create shareable definitions in a data definition language understood by many VAX programming language compilers and VAX Information Architecture products
- Store those definitions in the CDD database
- Modify those definitions in the dictionary without editing the programs and procedures that use the definitions
- Document the creation and use of the definitions in the dictionary
- Specify user access to individual definitions, based on thirteen separate access privileges and four user identification criteria

Programmers and other CDD users can:

- Copy definitions from the dictionary into programs at compile time
- Use VAX Information Architecture products to create CDD definitions automatically
- Document the use of a definition by making an entry in the definition's history list
- Maintain an area of the dictionary that contains data definitions for their private use

The CDD plays a crucial role in the VAX Information Architecture because it stores the data definitions used by VAX Information Architecture products, including:

- VAX ACMS application, menu, task group, and task definitions
- VAX DATATRIEVE domain, plot, record, table, and view definitions, and procedures
- VAX DBMS record definitions, schemas, subschemas, security schemas, and storage schemas
- VAX Rdb/VMS relation, constraint, index, view, and field definitions
- VAX TDMS form, request, and request library definitions

VAX programming languages can access CDD record definitions at compile time.

The VAX languages that can use the CDD are:

VAX COBOL Version 2.0 and later

VAX BASIC Version 2.0 and later

VAX PL/I Version 2.0 and later

VAX DIBOL Version 2.0 and later

VAX C Version 2.0 and later

VAX FORTRAN Version 4.0 and later

VAX PASCAL Version 3.0 and later

VAX RPGII Version 2.0 and later

In many cases, the same CDD record definition can be used unaltered by programs written in any of these VAX languages. Your definition can also specify special characteristics of a particular language compiler without affecting other compilers' use of the same definition.

2.2 Dictionary Organization

The CDD is organized as a hierarchy of dictionary **directories** and dictionary objects. Dictionary directories are similar to VMS directories: they organize information within the hierarchy. Dictionary **objects**, located at the ends of the branches in the hierarchy, are like the files in a VMS directory: they contain the data definitions stored in the dictionary. These definitions include:

- Record descriptions that can be copied into application programs
- Definitions required by VAX Information Architecture products

The CDD's hierarchical structure is like a family tree. Dictionary directories are the parents, and their children include other directories and dictionary objects. Figure 2-1 illustrates the hierarchical structure of the CDD.

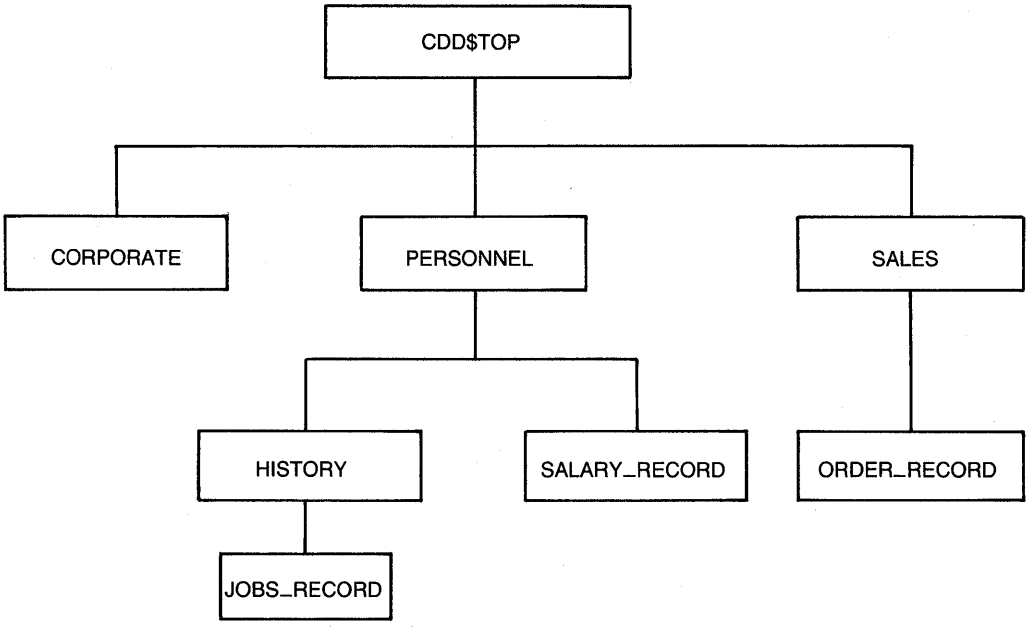
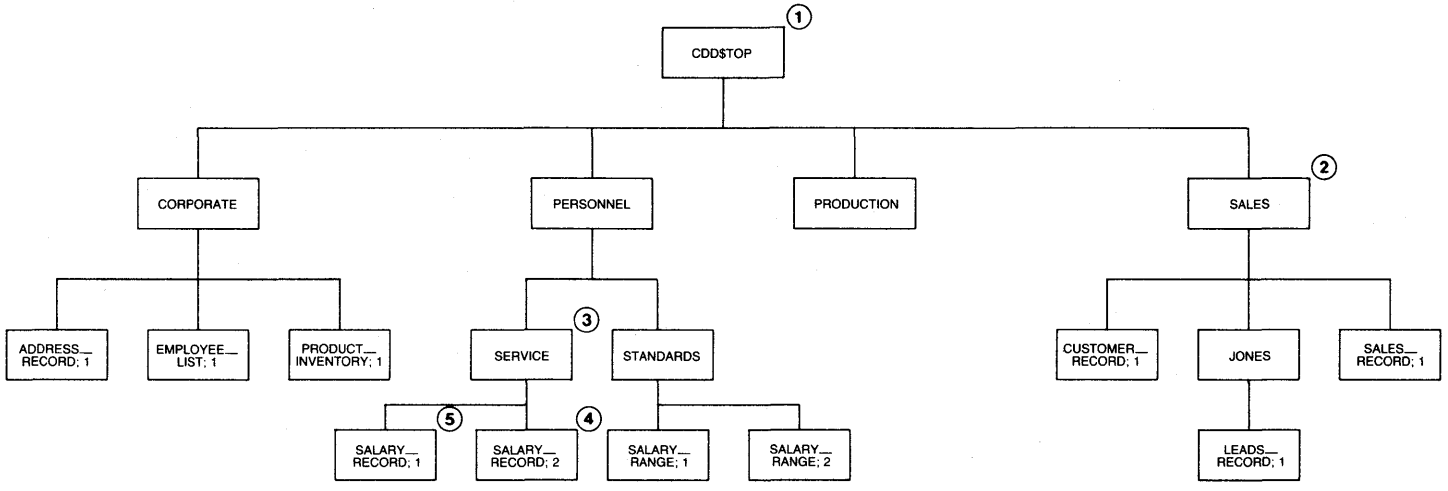


Figure 2-1: CDD Directory Hierarchy

The easiest way to understand the organization of the CDD is to look at the sample dictionary illustrated in Figure 2-2. It demonstrates the relationships that can exist between dictionary directories and objects. The sample dictionary is installed on your system as part of the CDD installation procedure; all the examples in the VAX CDD documentation set are drawn from this sample dictionary and its associated data definitions.

The numbers in Figure 2-2 correspond to the numbered explanations in the list following the figure.



MK-01575-00

Figure 2-2: Sample Dictionary

1. All directories and objects are descendants of CDD$TOP, the root dictionary directory. CDD$TOP is found at the top of the directory hierarchy and is created as part of the CDD installation procedure.

2. CORPORATE, PERSONNEL, PRODUCTION, and SALES are directories under CDD$TOP.

3. SERVICE and STANDARDS are directories under PERSONNEL. Similarly, JONES is a directory under the directory SALES. You can have any number of levels of directories under CDD$TOP.

4. SALARY_RECORD;2 is a dictionary object. It contains a record definition available to programs and information management products. Other dictionary objects in the sample dictionary are ADDRESS_RECORD;1, EMPLOYEE_LIST;1, PRODUCT_INVENTORY;1, SALARY_RECORD;1, SALARY_RANGE;1, SALARY_RANGE;2, CUSTOMER_RECORD;1 SALES_RECORD;1, and LEADS_RECORD;1.

5. It is possible to have multiple versions of the same dictionary object. SALARY_RECORD;1 is an earlier version of SALARY_RECORD;2. SALARY_RANGE;1 is an earlier version of SALARY_RANGE;2. Unlike VMS, however, the CDD does not create multiple versions of a record by default. For a more detailed explanation of multiple versions of dictionary objects, see *The VAX Common Data Dictionary User's Guide.*

Because the CDD is a directory hierarchy, different users can organize their portions of the dictionary according to their needs. This structure allows flexibility on several levels:

- Organizational

  Once the CDD is installed, each organizational unit can be assigned an independent portion of the dictionary. Each unit can use its portion of the dictionary without interference from the others.

- Departmental

  Different departments within organizations require shared access to some portions of the dictionary and independent access to others. The hierarchical structure of the CDD allows shared access.

Individual

Individuals can organize their own directories independently of other users, thus controlling access to sensitive material within their portions of the dictionary.

In the sample dictionary, for instance, the personnel, production, and sales departments all have separate portions of the dictionary. None has access to the data descriptions stored in the dictionary directories assigned to the others. However, they can all share the record definitions and documentary information stored in the CORPORATE directory.

Similarly, the definition stored in SALARY_RANGE;2 should be available throughout the whole personnel department, but not every section within the department needs access to the SALARY_RECORD;2 definition. The personnel department has solved this security problem by storing these record definitions in different directories with different access restrictions.

Finally, the CDD hierarchy allows users on the lowest levels to tailor the dictionary to their individual needs. Jones is a supervisor in the sales department and so has access to the department's record definitions and data descriptions. In addition, she has been assigned the directory JONES for her own use, and in it she has stored LEADS_RECORD;1, a record definition for data identifying potential customers.

To reach a target directory or object, you travel down a path from CDD$TOP to the target. You specify the path name of a target by entering the names of all the directories, starting with CDD$TOP and ending with the name of the target. You separate each name in the path name from the others by a period. Thus, CDD$TOP.PERSONNEL.STANDARDS.SALARY_RANGE;2, is the full path name of SALARY_RANGE;2 and CDD$TOP.SALES.JONES is the full path name of the directory JONES.

You can also use shorter forms of the path name, called relative path names and given names, to specify a directory or object. For more information about these forms of the path name, see the *VAX CDD User's Guide.*

2.3 CDD Features

Users of a data dictionary must be able to perform such essential tasks as:

- Creating and storing data definitions
- Controlling access to definitions
- Assessing the impact of changing data definitions
- Modifying existing definitions
- Locating the correct definition for an application program
- Copying the definition into application programs
- Maintaining the dictionary files

For more complete descriptions of these features, see the manuals in the CDD documentation, particularly the *VAX Common Data Dictionary User's Guide.*

2.3.1 Creating and Storing Data Definitions

The CDD Data Definition Language Utility (CDDL) provides a generic language that lets you define records for use by many VAX programming languages and by VAX Information Architecture products. CDDL also allows you to update these CDD record definitions.

Note that the CDD accepts definitions from all VAX Information Architecture products; however, these definitions are inserted into the CDD through the products' definition utilities instead of through CDDL.

To create a CDDL record definition and insert it into the dictionary, you:

- Create a CDDL source file using VAX EDT or some other text editor
- Submit the source file to the CDDL compiler, which inserts the record definition into the dictionary if the source file compiles without error

Figure 2-3 is a listing of EMPLOYEE.DDL, a typical CDDL source file.

```
DEFINE RECORD CDD$TOP.CORPORATE.EMPLOYEE_LIST

  DESCRIPTION IS

  /* This record contains the master list of all
     employees */.

    EMPLOYEE STRUCTURE.

    /* An employee's ID number is his
       or her social security number */

        ID                DATATYPE IS UNSIGNED NUMERIC
                          SIZE IS 9 DIGITS.

        NAME STRUCTURE.

          LAST_NAME       DATATYPE IS TEXT
                          SIZE IS 15 CHARACTERS.

          FIRST_NAME      DATATYPE IS TEXT
                          SIZE IS 10 CHARACTERS.

          MIDDLE_INITIAL  DATATYPE IS TEXT
                          SIZE IS 1 CHARACTER.

        END NAME STRUCTURE.

        ADDRESS             COPY FROM
                            CDD$TOP.CORPORATE.ADDRESS_RECORD.

        DEPT_CODE          DATATYPE IS UNSIGNED NUMERIC
                           SIZE IS 3 DIGITS.

    END EMPLOYEE STRUCTURE.

  END EMPLOYEE_LIST RECORD.
```

Figure 2-3: Listing of EMPLOYEE.DDL

The DEFINE statement names the record definition. The name you enter is the path name of the definition. The last name of the path name (in the example, EMPLOYEE_LIST) is called the **given name** of the definition; the rest of the path name is the path of directories to that object. Thus, you can name an object and specify its place in the dictionary in one step.

The DESCRIPTION statement documents a record definition. You can enter a DESCRIPTION statement to explain the entire record definition and any individual field in the record. EMPLOYEE.DDL contains an example of each type.

Field description statements describe the field characteristics of a record. They include the names and data types of the fields, as well as other information.

CDDL supports four different kinds of field statements:

- Elementary field description statements describe fields that are not divided into other fields. The ID field in EMPLOYEE.DDL is an example of an elementary field description statement.
- STRUCTURE field description statements describe fields that are divided into one or more subordinate fields. EMPLOYEE STRUCTURE is a STRUCTURE field description statement that includes all the other fields in EMPLOYEE.DDL as subordinate fields.
- COPY field description statements copy the contents of an existing record definition into a new record definition. The ADDRESS field of EMPLOYEE.DDL is an example of a COPY field description statement. When CDDL compiles EMPLOYEE.DDL, it copies the contents of another definition in the dictionary, called a template record, into the ADDRESS field. In Figure 2-3, the template record is CDD$TOP.CORPORATE.ADDRESS_RECORD. Thus, certain commonly used fields are defined in the same way in every record definition that uses them.
- VARIANTS field description statements provide alternative descriptions for the same portion of a record. VARIANTS are similar to the REDEFINES clause in VAX COBOL and VAX DATATRIEVE.

Field attribute clauses describe characteristics of the fields in a record. There are two types of field attribute clauses: general and facility-specific. General field attributes describe the storage of data definitions in the CDD. All language processors that use the CDD recognize these attributes. DATATYPE is an example of a general field attribute.

Facility-specific field attributes describe characteristics of a data definition that affect how a particular compiler or product uses it. Other languages and products that do not support an attribute ignore its facility-specific attribute clause. Thus, you can tailor a characteristic of a record definition to a particular language or product without making the definition unacceptable to others. For example, the following field definition contains a facility-specific attribute clause, BLANK WHEN ZERO, that is useful only to VAX COBOL:

```
ZIP_CODE STRUCTURE.

     NEW                    DATATYPE IS UNSIGNED NUMERIC
                            SIZE IS 4 DIGITS
                            BLANK WHEN ZERO.

     OLD                    DATATYPE IS UNSIGNED NUMERIC
                            SIZE IS 5 DIGITS.

END ZIP_CODE STRUCTURE.
```

When other compilers and products (like VAX PL/I or VAX DATATRIEVE) encounter this field definition, they ignore the BLANK WHEN ZERO clause.

You can find the source file containing every record definition in the sample dictionary in Appendix A of the *VAX Common Data Dictionary Data Definition Language Reference Manual.*

2.3.2 Controlling Access to Data Definitions

A major goal of a data dictionary is to let users share data definitions; however, you may want to keep some definitions confidential, and you certainly need to limit the number of people who can enter, change, or delete definitions in the dictionary. The CDD provides security mechanisms to protect the dictionary against browsing or modification by unauthorized users.

You control access to any dictionary directory or object through an access control list (ACL). When a user wants access to a particular directory or object, the CDD checks the item's access control list to determine that user's privileges.

The CDD also provides the Dictionary Management Utility (DMU) to create and manage the dictionary structure: it lets you copy, rename, and back up portions of the dictionary, to restore backed-up portions, to display the contents of the dictionary, and to document dictionary use. DMU also has commands that can grant or deny privileges to users.

2.3.3 Subdictionaries

The CDD installation procedure creates a file called CDD.DIC that can contain your entire dictionary. However, you can also create other dictionary files to hold portions of your dictionary. If you do so, DMU creates a directory in CDD.DIC that points to a separate dictionary file with the name you specify. The directory that points to that separate file is called a subdictionary directory or subdictionary.

A subdictionary file can be stored anywhere; it need not be stored on the same disk as CDD.DIC and it need not be accessible when other portions of the dictionary are in use.

Except for its physical location, a subdictionary is just like any other directory. A subdictionary is part of the same dictionary hierarchy and performs the same functions as dictionary directories. Most CDD users notice no difference between using a dictionary directory and using a subdictionary. Although they are part of a single dictionary, subdictionaries provide you with the benefits of having multiple dictionaries on a system.

Subdictionaries can be very helpful in certain situations:

- You can store sensitive material off line when it is not being used. For example, the record definitions used by the personnel department may be sensitive, so the data administrator can create a PERSONNEL subdictionary directory that is stored on a separate disk.
- Creating subdictionaries lets you use VMS file protection to control access to different dictionary files. The CDD has a protection system that lets you control access to individual directories and objects in the dictionary. This system protects you when you are using the CDD utilities, but it does not protect the dictionary file. VMS file protection provides another layer of protection to augment CDD access control lists.
- You can use one dictionary to serve several distinct organizations, as in a time-sharing system. Each organization can have its own subdictionary on its own disk. You can charge each organization for the amount of dictionary space its data descriptions use.

For more information about subdictionaries, see the *VAX Common Data Dictionary User's Guide.*

2.3.4 Tracking Changes to the CDD

The CDD gives you the capability of documenting the use of dictionary objects. For example, before modifying a record definition, the data administrator needs to know which other definitions are affected and which programs and procedures need to be changed as a result. Programmers using the dictionary also need to know at a glance the purpose and the contents of a definition they are considering copying into an application.

The CDD's history list feature lets you document and monitor the use of each dictionary directory, subdictionary, and object. This list of operations makes up an audit trail for each dictionary element. A history list entry contains information about an operation, including the action taken, the person responsible, the facility used, and the date and time. You can create an entry in a history list for a directory, subdictionary, or object when you:

- Create or modify a directory, subdictionary, or object
- Modify an access control list
- Copy a directory, subdictionary, or object to another part of the dictionary

- Access an object from a VAX programming language or a VAX Information Architecture product

For CDDL, DMU, and most of the languages and products that use the CDD, you use the /AUDIT qualifier to create a history list entry. You can add your own text to the information automatically stored in a history list entry.

These entries are a valuable aid to the person responsible for managing the dictionary. If history lists are maintained, the data administrator can assess the impact of changing a record definition or other dictionary object. For example, if a user wanted to change a record definition CDD$TOP.CORPORATE.ADDRESS_RECORD, she could first look at the history list for that definition and see which other record definitions and which application programs use it. If the definition changes, the history list shows which application programs and other definitions must also change.

2.3.5 Modifying Data Definitions

The information needs of organizations change, so it is often necessary to change data definitions as well. With CDDL, you can either:

- Replace an existing record definition with a new one without losing the existing access control list and history list
- Create a new version of a record definition, keep the old version as a backup, and copy the access control list and history list to the new version

CDDL/REPLACE replaces an existing record definition with a new one. All you need to do is:

- Create a new CDDL source file that contains the path name of the record definition you want to replace
- Compile the new source file with the command CDDL/REPLACE

The CDDL compiler processes the new source file and:

- Removes the original definition and replaces it with the new version
- Keeps the original access control list and history list
- Creates a new history list entry documenting the change

If you want to keep the old version as a backup, you can use the same source file with the /VERSION qualifier to the CDDL command instead of /REPLACE.

In this case, CDDL:

- Creates an additional new version of the record definition with a version number one higher than the current version
- Copies the access control list and history list of the old definition to the new version
- Creates a new entry in the new version's history list documenting the change

When you are confident of the success of the new record definition, you can remove any backup versions of it with the DMU PURGE command.

The /RECOMPILE qualifier is useful if you have modified a template record. Once you have examined the history list and determined which record definitions use a template record, you need only name the record definitions in the CDDL/RECOMPILE command. CDDL then recompiles them with the information in the modified template record. The /RECOMPILE qualifier deletes the old definition by default. If you want to save the old definition, you can use the /VERSION qualifier with /RECOMPILE.

2.3.6 Locating the Correct Data Definition

Programmers who want to copy record definitions need some way to find the definition they want. Using meaningful names for definitions is helpful, but even so, several definitions often have similar names. The CDD provides two methods for programmers to check the purpose and contents of a record definition:

- You can read explanatory text that was inserted into the CDD as part of the record definition source file. This text is always available to inform users of the purpose of that record definition.
- Alternatively, you can use the DMU LIST command with the /ITEM=SOURCE qualifier to see the source code for the record definition.

2.3.7 Copying the Definition into Application Programs

Once a programmer finds the desired definition, it can be easily included in the program at compile time. For example, in a VAX COBOL program, you use COBOL's COPY statement in the Data Division section of the program.

The COBOL compiler retrieves the definition from the CDD and compiles it as COBOL object code. If you use the /AUDIT qualifier when you compile the program, the COBOL compiler also makes an entry in the history list of the record definition to document the transaction.

The other VAX programming languages that use CDD record definitions work in a similar manner.

2.3.8 Maintaining the Dictionary

Dictionary files, like any other files, can become corrupted by hardware failures or other causes. The CDD Verify/Fix Utility (CDDV) lets you check the condition of your dictionary and subdictionary files and to repair them if they have been corrupted.

If disk space is a problem, CDDV also lets you compress dictionary and subdictionary files, returning free space to the operating system for use by other files.

No one can use a dictionary or subdictionary file while you are using CDDV on it, but users can work in the main dictionary file and other subdictionary files while you use CDDV on one subdictionary file. Only a user with VMS SYSPRV or BYPASS privilege can use CDDV in the root dictionary file, but users who own subdictionaries can use CDDV on any subdictionary file they own.

VAX Rdb/VMS 3

3.1 Overview of VAX Rdb/VMS

VAX Rdb/VMS is a relational database management system for VAX computers that use the VMS operating system. Rdb/VMS gives you the advantages of a full-featured database management system, including data security, integrity, and optimized access. Because Rdb/VMS uses the relational model of data storage, an Rdb/VMS database is flexible and easy to use. The relational model offers several advantages over other data models:

- The structure of the database is easy to understand and easy to explain to users.
- The database lets you combine and compare data in a wide variety of ways. You can define relationships between data dynamically.
- A programmer or analyst can create, modify, and maintain the database.

VAX Rdb/VMS lets you access the data in the database in several ways:

- Through VAX DATATRIEVE, DIGITAL's query and reporting language
- Through application programs
- Through RDO, a simple terminal interface language similar to DATATRIEVE

3.1.1 VAX Rdb/VMS as a Database Management System

VAX Rdb/VMS provides the features associated with a database management

system:

- Data independence and consistency

  You can remove data definitions from application programs and store them with the data. Because Rdb/VMS reduces the storing of redundant data, it helps ensure that updates do not result in inconsistent data. Rdb/VMS also lets you centralize the management of data definitions, both within the database file and within the VAX Common Data Dictionary. You can use views to bring together data stored in separate relations in the database.

- Interactive, multi-user environment

  More than one user can have access to the data at the same time, yet each user can work from a customized view that may include only a subset of the entire database. Rdb/VMS also ensures that one user's operations on the database do not lead to inaccurate results for other users.

- Data integrity and security

  Rdb/VMS maintains the integrity of the database in the event of user errors, hardware or software failures, and concurrent use of the database. Furthermore, the Rdb/VMS security mechanism prevents access to data by unauthorized users.

- Centralized administration

  A single user can handle the responsibilities of administering the database. This centralization of database administration tasks helps ensure the consistency of the database. The person responsible for managing the database uses a set of simple commands that make database maintenance and control easy.

3.1.2 When and Where to Use VAX Rdb/VMS

The choice of a data management product depends on many factors, including the size, number, and complexity of the data files involved, the capacity of the system, the number of concurrent users, and the types of operations users perform. The most important consideration is the suitability of the data model for the particular application. Some applications, especially those that involve complex relationships and large amounts of data, run best with a CODASYL-style system such as VAX DBMS.

In general, VAX Rdb/VMS is intended for use in applications that meet the following criteria:

- The database structure needs to be comprehensible to nonprofessionals.

  VAX Rdb/VMS uses the relational data model, which organizes data into tables called relations. Because people often see data represented in tables, even users without knowledge of database management systems can understand the organization of the data in a relational system. If people without professional knowledge of database management will use your database frequently, Rdb/VMS may be the logical choice.

- The structure of the database changes frequently.

  Rdb/VMS permits easy, interactive restructuring. As the needs of your organization change, you can add indexes, fields, and relations to the database. Similarly, you can delete outdated information. You can also build prototype systems to test the structure of your database without committing extensive resources to the effort.

- The system must provide a high degree of data independence.

  VAX Rdb/VMS bases relationships between data on the values stored in the database, not on predefined data structures. For this reason, your database query can dynamically define or redefine relationships between data.

- The database administrator's job is much easier.

  The database designer can translate a logical database design into a working database with a simple set of statements, entered interactively at the terminal or in a command file. If and when you need to restructure the database, you can do so with virtually no inconvenience to database users.

Because the relational model is easy to understand, Rdb/VMS is useful for quick implementation of simple applications. However, Rdb/VMS is also sophisticated enough to handle even complex database applications efficiently.

3.2 The Relational Model

In a relational database, data resides in tables known as **relations**. A relation consists of rows and columns. Where a row and column intersect, you can store no more than one data item. The following sections explain these concepts.

3.2.1 Relations

You have probably seen data presented in tabular form many times. A table consists of a set of rows and columns. The columns, which usually have names, divide each row into a set of fields. For a single field in a row, there can be only one piece of data. The absence of repeating groups and group fields simplifies the structure of the database and allows easy access to each data item.

Figure 3-1 is a representation of a typical Rdb/VMS relation that shows employee information.

Relation ——➤ Employees **Field ↓**

	FIRST_NAME	LAST_NAME	BIRTH_DATE	SOCIAL_SECURITY
	James	Adkins	11-MAR-1932	910509184
	Louie	Ames	13-APR-1941	330590912
	Ann	Andriola	25-JAN-1960	736305984
	Jo Ann	Augusta	30-MAY-1960	703845440
	Joseph	Babbin	12-DEC-1927	329016224
	Beverly	Barradas	8-JUN-1952	356251008
	Dean	Bartlett	5-MAR-1927	269212608
	Paul	Bellivea	9-MAY-1955	067822216
	Nancy	Bennett	14-FEB-1955	049893260
Record ——➤	Nancy	Brown	7-OCT-1942	818547968

Figure 3-1: A Relational Table

In this relation, each field represents an item of data for each employee. Each record represents the data for a single employee. To find the data stored in any location of the relation, you need only name the relation and specify the intersection of field and record.

To find Nancy Brown's social security number, for example, ask Rdb/VMS the question like this:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = "Brown" AND
  E.FIRST_NAME = "Nancy"
    PRINT
      E.FIRST_NAME,
      E.LAST_NAME,
      E.SOCIAL_SECURITY
END_FOR
```

To find the answer, Rdb/VMS:

- Finds the record in EMPLOYEES in which the LAST_NAME field is occupied by the name "Brown" and the FIRST_NAME field is occupied by the name "Nancy"
- Finds the intersection of that record and the SOCIAL_SECURITY field
- Displays on the terminal screen the contents of the three fields named in the statement

E IN EMPLOYEES declares a **context variable**. This variable lets you refer to the EMPLOYEES relation unambiguously within the PRINT statement. The result of the present query is:

```
Nancy          Brown          818547968
```

The concept of the relation distinguishes the relational model from the two other most frequently used database models: hierarchical and network.

A hierarchical database organizes the relationships between record types in a tree structure. It stores related records on the same branch of the tree to make data retrieval efficient.

A network database uses sets to establish relationships between records. A single record can participate in any number of sets, so you can relate it to any other record in the database, not only to those above and below it in a hierarchy. This arrangement allows flexibility in setting up data relationships to match the needs of the organization.

Both the hierarchical and network models assume that you know how data is related. Relationships are preestablished and difficult to change once the database is in use. In contrast, you can define relationships dynamically in a relational database by relating a field in one relation with a field in another. Thus a relational database gives greater flexibility in setting up relationships and allows for easier restructuring than either of the other two models. On the other hand, because relationships are not part of the database's physical definition, data retrieval may be slower in complex or very large applications.

3.2.2 Normalization

If you think for a moment about the EMPLOYEES relation discussed previously, you can imagine many other kinds of information to store there. For example, you might want to keep track of the job history of the employee, including all previous jobs and their starting and ending dates. However, there is no way to represent repeating groups in an Rdb/VMS relation; only one data item can occupy an intersection. Therefore, to store information about five previous jobs for an employee,

you would also have to repeat the name, address, identification number, and other employee information five times. Then there would no longer be a one-to-one correspondence between records in the relation and employees in the company.

You might also want to include in the EMPLOYEES relation information about each job an employee has held, such as the salary range associated with that job. If you store job information in the EMPLOYEES relation, however, you would have to store information about a particular job category many times, once for each employee who held that job.

Storing all the information that might be relevant to employees in one relation, then, leads to a great deal of redundancy -- the same data stored in more than one place. Redundancy of data has two disadvantages:

- It wastes space in the database.
- It makes updating information difficult. If you store the salary range for a particular job code with each employee in the EMPLOYEES relation, you must find and change all the occurrences whenever the salary ranges change. If you miss some, the database is no longer consistent.

A process known as **normalization** solves these two problems. Normalization ensures that the database keeps separate concepts physically separate. Thus you store a data item only once, and you need to perform only one update operation to change it. When you need to bring data together from different relations -- when you want an employee's job history, for instance -- the database lets you create temporary relationships by joining relations together. VAX Rdb/VMS works best with well-designed, normalized databases. To learn more about the process of normalization, see the *VAX Rdb/VMS Guide to Database Design and Definition.*

The next section describes the relational join and other operations characteristic of a relational database.

3.2.3 Relational Operations

The main advantage of a relational database is the ease with which you can retrieve precisely the information you want, even if you must gather the information from data stored in several relations.

3.2.3.1 Joining Relations -- Once you have normalized your data, data items that are not directly related to each other may be separated into different relations. To establish relationships between such data items, you need to bring those separate relations together with a join operation. The **relational join** selects a record from one relation, associates it with a record from another relation, and presents them as though they were part of a single record. The join is sometimes referred to as a **cross operation.**

The simplest form of join is called a cross product. The **cross product** associates each record in one relation with each record in another relation. This kind of join retrieves all the records in both relations, repeated many times, which is usually far more data than you need.

A more useful type of join involves matching values in a field from one relation with those in a corresponding field in another relation. This is sometimes called an **equijoin**. For instance, two relations may contain a field described as a five-digit employee identification number. You can combine data from these two relations by matching the values in the common field. This type of join lets you establish relationships between data items in your database. If you have set up the database correctly, you can relate an item in any relation with items in any other relation.

For example, a second relation in the PERSONNEL database might contain information about the departments in the corporation. This second relation has three fields, DEPARTMENT_CODE, DEPARTMENT_NAME, and MANAGER_ID:

```
RDO>   FOR D IN DEPARTMENTS
cont>    PRINT D.DEPARTMENT_CODE,
cont>          D.DEPARTMENT_NAME,
cont>          D.MANAGER_ID
cont>  END_FOR

ADMN  Corporate Administration          00225
ELEL  Electronics Engineering           00397
ELGS  Large Systems Engineering         00369
ELMC  Mechanical Engineering            00215
ENG   Engineering                       00435
MBMF  Board Manufacturing               00287
MBMN  Board Manufacturing North         00248
MBMS  Board Manufacturing South         00341
MCBM  Cabinet & Frame Manufacturing     00405
                    .
                    .
                    .
```

Assume that you want a report that includes the names of the employees and the names of the departments in which they work. The department name is in the DEPARTMENTS relation and the employee's name is in the EMPLOYEES relation. To complete the report, you must take each EMPLOYEES record, find the corresponding DEPARTMENTS record by matching the values in the two DEPARTMENT_CODE fields, and attach the two records. This join associates each employee with the name of a department. To create this kind of report, you name the two relations and the common field; Rdb/VMS does the rest. The result looks like a single record; the common field, DEPARTMENT_CODE, appears only once.

Figure 3-2 illustrates a join operation.

```
FOR  E  IN    EMPLOYEES CROSS
     D  IN    DEPARTMENTS OVER DEPARTMENT_CODE
     PRINT    E.LAST_NAME,
              E.FIRST_NAME,
              E.DEPARTMENT_CODE,
              D.DEPARTMENT_NAME
END_FOR
```

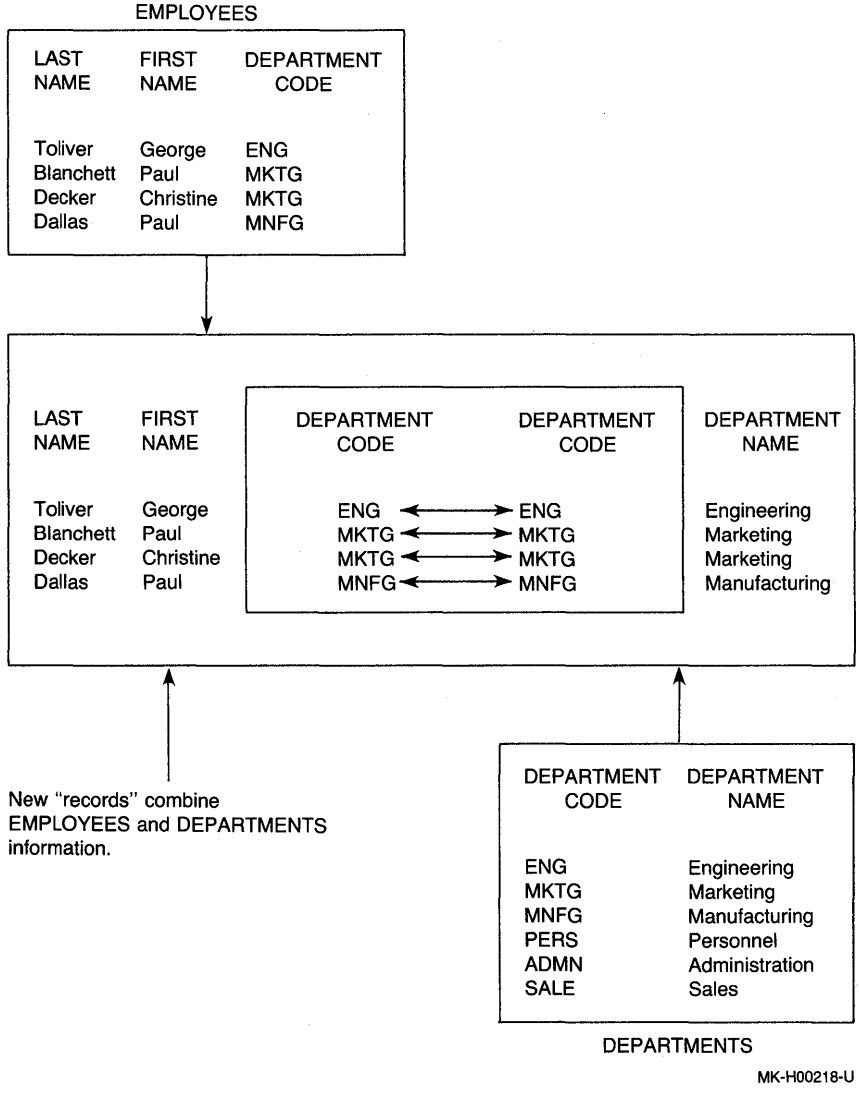


Figure 3-2: A Relational Join Operation

Joining two relations by matching values in a common field is the most frequent kind of join operation. However, you can also join relations using inequalities. For example, if you want to print the names of all employees whose salary exceeds the maximum for their job categories, the Rdb/VMS statement looks like this:

```
RDO> FOR E IN EMPLOYEES
cont>     J IN JOBS OVER JOB_CODE
cont>       WITH E.SALARY_AMOUNT > J.MAXIMUM_SALARY
cont>      PRINT
cont>        E.LAST_NAME,
cont>        J.JOB_CODE,
cont>        J.MAXIMUM_SALARY,
cont>        E.SALARY_AMOUNT
cont> END_FOR
```

The CROSS clause and the WITH clause set up two conditions for the join:

- The CROSS clause sets up the match between the EMPLOYEES relation and the JOBS relation. This cross associates each employee with the information on his or her current job.
- The WITH clause filters out all those records where the employee's salary falls within the prescribed range for the job.

3.2.3.2 Selecting Fields and Records -- In most cases, you do not want to retrieve all the records in a relation. You use a **record selection expression** (RSE) to select only the records you want.

An RSE is a clause in a data manipulation statement that names a set of conditions. An example of an RSE is the WITH clause in the previous example. When the statement containing the RSE executes, it creates a **record stream** made up of those records from the relation that satisfy the conditions set up by the RSE. That is, the RSE filters out everything but the records that contain the information you want to see. The RSE is at the heart of VAX Rdb/VMS because it lets you limit the record stream in many ways.

For a more detailed discussion of RSEs, see the *VAX Rdb/VMS Guide to Data Manipulation.*

3.2.3.3 Reducing Data to Unique Values -- A relational database lets you isolate unique values for a field. That is, you can establish a record stream that consists of all the records that contain a given value for a single field, with the repeated values removed.

For example, the EMPLOYEES relation contains a field for the supervisor's identification number. Assume that you want to compile a list of supervisors. The REDUCED TO clause lets you make a list that includes the values in the SUPERVISOR_ID field, listing each supervisor once. The clause filters out repetition of the supervisor ID numbers.

This process is sometimes called the **project operation**. In Rdb/VMS, the REDUCED TO clause performs the operation. It reduces the record stream to unique values of the specified field.

Once you have reduced a field to its unique values, you can combine the project and join operations to group records. You can also perform statistical functions on the groups of records you isolate.

3.2.4 Creating Views

You can create a logical structure that consists of a subset of records and fields from one or more relations in the database. To the user, this view looks exactly like a relation, even though no data is actually stored in a view.

A view can include any combination of fields and records from a single relation or from different relations. Views are most useful for making the result of a selection or join look like a relation. Normalized data is often separated into many relations and must be joined together to be useful. Once you have determined how to pull together information from multiple relations with a join operation, you can define a view to include that join. When users refer to the view by name, the join executes automatically. You can then display and manipulate information from multiple relations (with some restrictions) as though the relations were one.

3.3 Additional Features of VAX Rdb/VMS

In addition to the various features of a database management system and the simplicity of the relational model, VAX Rdb/VMS also provides many features to make the system easy to learn and easy to use. These features are included in the main interfaces to Rdb/VMS, the RDO utility and the program interfaces.

3.3.1 RDO, the Interactive Rdb/VMS Utility

VAX Rdb/VMS provides the Relational Database Operator (RDO), a single interactive environment for maintaining the database, creating and modifying definitions of database elements, and storing and manipulating data. When you enter RDO statements, Rdb/VMS executes those statements immediately.

RDO includes several types of statements:

- Statements for defining database elements

  The data definition statements of RDO (DEFINE, CHANGE, and DELETE) describe the entities of the database. You can define an entire database in one RDO session, including all the fields of all the relations, views, indexes, constraints on values, and protection. Later, you can change these definitions dynamically by adding or deleting fields or by changing the database characteristics you defined before. Therefore, restructuring the database is easy. You can delete database elements just as easily.

- Interactive data manipulation statements

  RDO includes data manipulation statements for learning and testing. These statements let you practice retrieving, storing, and modifying data. When you are ready to end the interactive session, you have the option either of making the changes you entered permanent by issuing a COMMIT statement or of removing the changes you entered by issuing a ROLLBACK statement.

  Application programs use the same data manipulation statements. Thus, you can use RDO to learn the principles of database management and to practice structuring statements. Then you can test the statements before including them in programs. This ability to test Rdb/VMS statements interactively lets you debug Rdb/VMS queries before compiling and debugging a program that uses them.

  Although you can use RDO as a query language for interactive retrieval of data, it is not designed for this purpose. VAX DATATRIEVE is more versatile than RDO for interactive use.

- Ease-of-use features

  RDO includes the SET and EDIT statements, which let you control your terminal session, and the SHOW and HELP statements, which give information about Rdb/VMS to interactive users.

- Utility statements for database maintenance

  You can also use RDO statements to maintain the database by:

  - Analyzing the database for space usage
  - Creating a backup copy of the database
  - Initiating database journaling and recovering transactions from the journal

3.3.2 Program Interfaces

There are two ways to use VAX Rdb/VMS from a program:

- Precompilers

  Rdb/VMS provides precompilers for several VAX languages. When a precompiler is available for your language, you can include Rdb/VMS statements in your program in almost the same form as they appear in RDO. You can refer to program variables in Rdb/VMS statements. Precompilers are available for the following VAX high-level languages:

  VAX BASIC
  VAX COBOL
  VAX FORTRAN
  VAX PASCAL

- Callable RDO

  For languages not supported by precompilers, Rdb/VMS provides the Callable RDO utility. This utility is an interpretive call interface, a single external routine that accepts an Rdb/VMS statement as a parameter. You can call this routine from any language that conforms to the VAX Procedure Calling Standard.

3.3.3 Multiple Databases and Remote Access

VAX Rdb/VMS lets you combine data from more than one database. **Database handles** allow you to give a temporary name to each database you want to invoke so that you can specify which database each relation comes from.

Similarly, you can access databases on remote nodes of a DECnet network. The database on the remote node can be either a VAX Rdb/VMS database or a VAX Rdb/ELN database. To access a remote database, you simply add the node name to the file specification.

3.4 Designing a Database

The size and complexity of your organization determine how difficult it is to design a database that meets the organization's information needs. If you are responsible for data processing in a single department or a small company, you may be able to design a database yourself quickly and easily. However, if you are trying to solve information management problems for a large organization with several departments and several types of business activity, database design is a much more complicated and time-consuming process.

Many books describe systems for designing a database. The *VAX Rdb/VMS Guide to Database Design and Definition* recommends one method. Whatever system you choose and whatever the size of your organization or its data management needs, the goal of the design phase is the same: to define a set of logical relations that models the data needs of your organization. Generating these logical relations requires at least these steps:

- Analyze the data to determine all the data items needed by all the members of your organization.
- Arrange data items into conceptual groups. For example, personnel information falls into categories such as employee information, department information, and job information.
- Simplify the groups by eliminating redundant information. As much as possible, find items that you need for more than one purpose and put them in one place. For example, if there are certain skills associated with a certain job, group those skills with a job code, not with information about individual employees. This is the first step in normalization.
- Remove repeating groups to separate relations. Because a relational database cannot have repeating groups, each field in each record can contain only one data item. The removal of repeating groups is the second step in normalization.
- Determine index keys for relations. An **index** keeps track of the location of each record in a relation so that the database system can find records directly, without scanning. An index uses a field or combination of fields in the record as a key.

  You should decide which fields you plan to use most often for data retrieval and join operations and make these key fields. Normally, you choose the field or combination of fields (called a multisegment key) that uniquely identifies records in the relation, such as the EMPLOYEE_ID field in the EMPLOYEES relation. As you use the database, you may find that you need more index keys or that some you have defined are unnecessary. You can add or delete indexes at any time.

  Using index keys allows Rdb/VMS to speed up data retrieval in many ways. For example:

  - Rdb/VMS uses indexes to optimize data access on single relations. If you retrieve employee information often by employee identification number, make that number a key to the employee relation.
  - Rdb/VMS uses indexes to make joins faster. As previously explained,

a join operation often involves the crossing of two relations over a common field. If that field is also a key in both relations, the join is faster. Determine which relations you plan to join frequently, and make the common field a key.

- Specify the relationships between data items by setting up common fields between relations. For example, you might characterize the relationship between "employee" and "department" as "is a member of." You might express this relationship in the logical model by including the department code as a field in the employee relation.

- Define views that correspond to those parts of the database that you have separated for efficiency and clear structure but that must be reunited if you are to extract information from them. For example, you may need to create a report that combines employee information with department information. You can define a view that includes a join operation to bring the information together.

- Establish constraints for each field to limit the valid entries for that field. You can specify two types of constraints in VAX Rdb/VMS:

  - You can limit the values of that field to a certain set when you define the characteristics of a field. For example, you may want to restrict the sex field in the employee relation to "M" or "F."

  - You can define a formal constraint for the field. This constraint can use other values in the database to check the field's value. For example, you can check a department code against the department relation to make sure the department code really exists before the code is entered in the job history relation.

- Specify privileges if you want to deny some users access to parts of the database. You can specify privileges for the database and for each relation in it. You can also specify privileges for views.

When you finish the design phase, you have a logical model for the database. This model specifies all the relations and data items you need. It also specifies the relationships between the relations, though you may want to dynamically specify more of these relationships at a later time. The model also points to fields in each relation that might be made into index keys in the physical database. Using views, you can make some of these relationships explicit and permanent.

The features of VAX Rdb/VMS make restructuring a database easy and quick. However, Rdb/VMS is designed to be accessed from application programs. If you restructure your database radically, you may also have to rewrite or recompile programs to take new structures into account. Furthermore, any restructuring of a database consumes time and system resources. No matter what type of data

management product you use, careful analysis and planning when the database is first set up saves maintenance time in the long run. It is especially important to normalize the database, as Section 3.2.2 and the *VAX Rdb/VMS Guide to Database Design and Definition* explain.

3.5 Database Operations

This section introduces the operations you use to create a database or to change an existing database. The process assumes that you have completed a design for your database and normalized the logical relations. You perform these operations with RDO, the interactive interface to Rdb/VMS.

You set up a database and its components with DEFINE statements. Each DEFINE statement enters a definition in the database file and, optionally, in the VAX Common Data Dictionary. Rdb/VMS can store definitions in two places for security and efficiency. Because the database file contains the definitions, the system can read them internally without searching the CDD each time a program runs. Because the definitions can also be stored in the CDD, other products, such as DATATRIEVE and high-level language compilers, can copy definitions from the CDD. You use the DEFINE statement to create the following items:

- Database

  When you define a database, RDO creates a database file, a snapshot file (which provides temporary storage for read-only data retrieval), and an entry for the database in the CDD.

- Relation

  The definition of a relation includes all of the relation's field definitions. Rdb/VMS also creates a default access control list for the newly created relation.

- Field

  A field definition specifies the type of data in the field and can be used in any relation definition.

- Constraint

  A constraint definition is the set of conditions that restrict the values in a relation.

- Index

  An index definition names a field or set of fields as a single or multisegment index key for a relation.

- Protection

  A protection definition creates an access control list entry for a database, a view, or a relation. An access control list entry contains a user identifier and a list of access rights granted to that user.

- View

  A view definition logically associates fields from one or more relations.

You can use RDO's DELETE statement to remove any of the items created with the DEFINE statement. In addition, you can use RDO's CHANGE statement to modify an entire database, a relation, a field, or a view.

3.6 Storing Data

There are three ways to store data into an Rdb/VMS database. You can enter data initially using:

- The STORE statement in RDO
- A program to read a data file and store the data
- VAX DATATRIEVE's restructure mechanism

Thus, to load a database from VAX DBMS into VAX Rdb/VMS, you use the DBMS UNLOAD utility to create RMS files for each record type in the DBMS database. You then use DATATRIEVE to turn each record type into an Rdb/VMS relation. Because some DBMS databases use implicit relationships, you may need to add new fields to some Rdb/VMS relations after loading in order to make these relationships explicit.

3.7 Accessing Data

Once you have defined the database and loaded the data, you can begin to retrieve and manipulate data. There are several ways to access the data in an Rdb/VMS database:

- Through RDO, the terminal interface to Rdb/VMS
- Through VAX DATATRIEVE
- Through programs, using embedded data manipulation statements
- Through programs, using Callable RDO

Rdb/VMS is optimized for use by programs, and users will most often access an Rdb/VMS database through application programs and DATATRIEVE. RDO is intended as an interactive environment for learning, testing, and prototyping. RDO data manipulation syntax is virtually identical to the syntax you include in programs. Thus you can test queries and update operations interactively in RDO and include the statements directly in programs.

3.7.1 Transactions

When you include Rdb/VMS statements in an application program, you need to be aware of some additional features of Rdb/VMS. Rdb/VMS supports the concepts of transactions and record locking. These features help ensure the consistency and accuracy of the retrieved data when two or more programs are retrieving and updating data at the same time.

A **transaction** is a series of operations that must execute as a unit or not at all. The use of transactions ensures that operations on the database are never partially completed.

For example, when an employee is promoted, you may run a program to change the employee's job code and salary. The steps in the program follow:

- The program adds records for the employee's new job to a JOB_HISTORY relation.
- The program adds records for the employee's new job to a SALARY_HISTORY relation.
- The salary figure entered in the SALARY_AMOUNT field of SALARY_HISTORY exceeds the maximum salary for the new job code. (Remember, you can define constraints for each field in a relation.) Rdb/VMS returns an error message and does not modify the SALARY_HISTORY record.
- At this point, the database is inconsistent; a new job record has been entered into the JOB_HISTORY relation but not to the SALARY_HISTORY relation.

To prevent such problems and to ensure consistency, Rdb/VMS lets you group such update operations in a single transaction. You commit or roll back that transaction as a unit. A program does this in the following way:

- The program includes all the update operations in one transaction before beginning the update.
- The program tests for problems (such as the validity problem mentioned previously) in each part of the transaction.

- If all the updates are successful, the program commits the transaction by entering the changes in the physical database file.
- If any of the updates are not successful, the program rolls back the entire transaction and no update takes place.

Hardware or system failures can also interrupt Rdb/VMS transactions. In such cases, Rdb/VMS does an automatic rollback.

3.7.2 Ensuring Consistency

Rdb/VMS lets many users concurrently read, write, and modify data in the database. However, this feature also introduces the possibility of inconsistency. For example, if two users read and then modify a single field, one of them may be reading an obsolete value. In general, a program that updates a value must be sure that the update will be complete before any other program updates the same value.

Rdb/VMS ensures consistency by allowing your program to protect relations and records from actions by other programs during a transaction. When you start a transaction, you can include a list of relations and the kind of access your program allows to other programs. For example, if your program is updating a relation, it might start a transaction with PROTECTED WRITE access. Such access allows other programs to read the relation, but no other program can write data there until your operations complete by either committing or rolling back the transaction.

3.7.3 Read-Only Transactions (Snapshots)

A transaction can also specify READ_ONLY access, which lets you take a snapshot of the database. In this mode, you can retrieve data from a relation without locking other users out of the database. You see a version of the data that is correct as of the moment the transaction starts. For simple reports, where the most up-to-the-minute information is not vital, READ_ONLY mode allows fast performance and a minimum of locking conflicts.

3.8 Retrieving Data

Retrieving data from an Rdb/VMS database is a simple and straightforward operation. You establish a record stream (with either a FOR statement or a START_STREAM statement) and specify which records are to be retrieved (with a record selection expression). You can use a SORTED BY clause to specify the order in which records are to be retrieved. You then determine which fields you want to retrieve and either display them with the PRINT statement or assign them to program variables with the GET statement.

3.8.1 Record Selection Expressions (RSE)

The record selection expression (RSE) defines specific conditions that individual records must meet before Rdb/VMS includes them in a record stream.

The following list shows the clauses of the RSE and the operations they perform:

- FIRST n

  Retrieves only the number of records specified. This clause normally accompanies the SORTED BY clause, because there is no guarantee of sort order in record streams.

- WITH

  Names a set of criteria for selection, using conditional expressions.

- CROSS

  Names another relation for a join operation.

- SORTED BY

  Names a key field by which to sort the record stream.

- REDUCED TO

  Names one or more fields to serve as the reduce key. The record stream consists of the unique values for that field or fields.

Using these clauses, alone and in combination, you can limit the record stream to exactly the data you want, and you can combine related fields from many relations in the database.

RSEs can also contain conditional expressions. A conditional expression represents the relationship between two field values. The value of a conditional expression is either true or false. Rdb/VMS **relational operators** specify the type of comparison to perform on the pair of field values. They are:

EQ =
NE <>
GT >
GE >=
LT <
LE <=

In addition, **logical operators** can link together multiple conditional expressions. The Rdb/VMS logical operators are AND, OR, and NOT.

You use conditional expressions most often as the object of the WITH clause in the record selection expression. By linking value expressions with relational operators and linking conditional expressions with logical operators, you can specify exactly the data you want to retrieve.

For example, to display the names of all the employees who live in Massachusetts and work in the Engineering Department or the Manufacturing Department, enter the following:

```
FOR E IN EMPLOYEES
  CROSS JH IN JOB_HISTORY OVER EMPLOYEE_ID
  CROSS D IN DEPARTMENTS OVER DEPARTMENT_CODE
    WITH E.STATE = "MA"
    AND D.DEPARTMENT_NAME = "Engineering"
    OR D.DEPARTMENT_NAME = "Manufacturing"
      PRINT E.LAST_NAME,
            E.FIRST_NAME
END_FOR
```

Rdb/VMS can also display statistical expressions based on values in the database. Table 3-1 lists the statistical expressions available.

Table 3-1: Statistical Expressions

Statistical Expression	Result
AVERAGE	Average of nonmissing field values in current stream
COUNT	Number of records in current stream
MAX	Largest value of field in current stream
MIN	Smallest value of field in current stream
TOTAL	Sum of values of field in current stream

3.8.2 Record Streams

Rdb/VMS provides two ways of determining the subset of records, called a record stream, to be retrieved from the database.

In most cases, your program establishes a record stream with a FOR statement.

The beginning of a typical FOR statement looks like this:

```
FOR E IN EMPLOYEES WITH E.STATE = "MA"
```

The result of this FOR statement is a record stream consisting of all the EMPLOYEES records with the string "MA" in the STATE field.

Once you have established a record stream with a FOR statement, the PRINT statement retrieves a specified set of fields from each record in the stream, one record after another.

A FOR statement works well when you want to process the records from a single record stream one at a time. There may be times, however, when you want to establish more than one record stream and want the processing of the streams to interact. In such cases, you can use a START_STREAM statement to start each stream. After you set up a record stream with the START_STREAM statement, you must use a FETCH statement to make each successive record in the stream available for processing.

FOR loops are easier to use than the START_STREAM statement. START_STREAM, however, gives your program more flexibility. For example, you can start more than one record stream, and the values returned from one stream can affect the processing of the other.

3.9 Modifying Data

Rdb/VMS lets you store, modify, and erase data using the same kind of record selection expression you use to retrieve data. Thus you can precisely specify the records you want to change. The following list summarizes the Rdb/VMS statements that modify data:

- STORE

  Stores values in the database

- MODIFY

  Modifies values in the database

- ERASE

  Deletes records from relations

3.10 Maintaining a Database

RDO provides a set of utility statements so you can perform common database maintenance functions, such as backing up and restoring data, analyzing space usage, checking database integrity, and maintaining journal files to restore a database if there is a failure.

- Analyzing space usage

  The ANALYZE statement displays the space usage for the database file. Optional qualifiers display the number of records and the index structure for each relation within the database. Regular analysis of database usage lets you restructure your database to improve processing efficiency.

- Saving a copy of the database

  Normally, you use regular VMS utilities, such as COPY and BACKUP, to save copies of the database for security against catastrophe. Rdb/VMS also provides BACKUP and RESTORE statements that let you create a copy of the database that you can restore on a compatible Rdb database system, such as VAX Rdb/ELN.

- Using journal files

  VAX Rdb/VMS keeps two kinds of journal files:

  - A before-image journal, which keeps a record of transactions in progress. In case of an error in a program, a system failure, or a user's ROLLBACK statement, the system can undo the changes made by the transaction. Before-image journaling is done automatically by the database system.

  - An after-image journal, which keeps a record of changes made to the database by committed transactions. You can use an after-image journal to rebuild a database that has been corrupted by a hardware or software failure. You can enable and disable after-image journaling. In case of failure, you can use the RECOVER statement to apply a journal file to a backed-up copy of the database.

3.11 Types of VAX Rdb/VMS Product Kits

VAX Rdb/VMS provides three kits:

- VAX Rdb/VMS

  VAX Rdb/VMS is the full development kit and contains all the components needed to create and use Rdb/VMS databases.

- VAX Rdb/VMS RUN-TIME

  VAX Rdb/VMS RUN-TIME is the run-time kit for VAX Rdb/VMS. It lets you use Rdb/VMS databases built with the full development kit but does not let you create databases.

- VAX Rdb/VMS REMOTE

  This kit contains all the Rdb/VMS components needed to access a full Rdb/VMS database system on a remote node.

VAX DBMS 4

4.1 Overview of VAX DBMS

A **database** is an organized collection of stored information that lets you separate the description of the data from the programs that use it. VAX DBMS is a multi-user, general-purpose, CODASYL-compliant database management system that runs on the VMS operating system.

A **database management system** increases the productivity of your application development effort by letting you divide the overall task into groups of logically related functions:

- Database administration

  Includes the design, creation, and maintenance of the logical database structure and the physical database management system. It also provides for the integrity and security of the data in the database.

- Application programming

  Includes the design, implementation, and maintenance of the programs that are the primary method of database access for the user. A database management system allows programmers to focus on the data stored in the database instead of on the physical representation of that data.

You can use VAX DBMS to access and administer databases ranging in complexity from simple hierarchies to complex networks with multilevel relationships. VAX DBMS supports full concurrent access in a multi-user environment without compromising the integrity and security of user data.

Ever since the idea of database management originated in the early 1960s, CODASYL (the Conference on Data Systems Languages) has been active in developing specifications for database management systems. Throughout the

1970s, the conference published several reports outlining the requirements for such systems. VAX DBMS is based on the March 1981 Working Document of the ANSI Data Definition Language Committee.

VAX DBMS is designed for users working in a structured application environment. Such users include programmers, analysts, designers, or administrators who use conventional planning and coding techniques to design, create, and maintain long-term applications for corporate use. The CODASYL principles that guided the development of VAX DBMS provide several benefits to such users:

- Data independence

  Data definitions are removed from application programs and centralized in the VAX Common Data Dictionary (CDD). The same data definitions can be used in high-level language programs and in VAX DATATRIEVE. Relationships among records can be defined in terms of sets, but the physical characteristics of these records and sets remain separate from the data definitions.

  VAX DBMS provides a data definition language (DDL) that lets you design a database that is as simple or as intricate as your applications require.

- Consistent multi-user environment

  Many users have concurrent access to the data, yet each user is shielded from the effects of other users. Each user's program sees only data that has been committed, that is, updated in a correct and consistent manner. A program cannot see data that has been incompletely or improperly updated.

  A **transaction** is a series of operations that must execute as a unit or not at all. The use of transactions ensures that operations on the database are never partially completed. All user access to a VAX DBMS database is transaction-oriented. If an update transaction is not successfully completed, the Database Control System (DBCS) returns, or **rolls back**, the database to a condition identical to the one before the start of the transaction. In addition, the DBCS uses various locking techniques to let you restrict access to update data or even process-sensitive retrieval data to only one user at a time.

- Data integrity

  The integrity of the database is maintained in the event of user errors and hardware or software failures.

- Programmer productivity

  Administration of the database is centralized in the role of the database administrator (DBA). In addition to providing controlled allocation of the

database and improved maintainability and security, this central control improves application programmer efficiency. For example, the database programmer can access data without designing separate files for specific applications. Data definitions are copied into a program from the CDD when the program is compiled. Because programmers need to be concerned only with application logic, application programs become easier to write and debug.

Programmers use the VAX DBMS data manipulation language (DML) to access a database. DML is understood by the VAX COBOL language and is available to FORTRAN programmers through FORTRAN/DML, a VAX DBMS preprocessor to the VAX FORTRAN compiler.

VAX DBMS also has a precompiler that lets you insert DML statements into programs written in the following languages:

VAX BASIC
VAX BLISS
VAX C
VAX DIBOL
VAX PASCAL
VAX PL/I
VAX Ada

All other programming languages that conform to the VAX calling standard can use DML through the callable Database Query (DBQ) interface.

4.2 Enhancements to the CODASYL Model

VAX DBMS provides many features that expand the capabilities of the database administrator and the database programmer beyond the scope of a CODASYL database management system. These features provide a database management system that is more complete, easier to use, and more secure.

These features provide enhancements in the following areas:

- Database administrator (DBA) productivity
- Programmer productivity
- System performance
- Security
- DECnet network access
- Operation in a VAXcluster

The following sections describe the features that relate to these areas.

4.2.1 DBA Productivity

Several VAX DBMS features and tools are available to increase the productivity of the database administrator:

- Database Operator (DBO) utility

  - Online database verification in CONCURRENT, PROTECTED, and EXCLUSIVE modes
  - Offline full and incremental database backup
  - Restoration (roll forward) of committed transactions to a backed-up database using the after-image journal
  - Simple restructuring of a database without unloading and loading all the data

- Load/Unload facility

  This facility allows a DBA to load VAX RMS records into an existing database. It can also be used to restructure a database by unloading and then reloading the database's records.

- After-image journaling

  An after-image journal contains only valid (that is, committed) changes made to a database. This feature lets you restore your database in the event of a storage device or system software failures.

- DDL compiler

  The DBA can use the DDL compiler to create a default storage schema, subschema, and security schema from a schema. By default, the DDL compiler automatically stores successfully compiled or generated definitions in the VAX CDD.

4.2.2 Programmer Productivity

Programmers must make many choices in the course of developing application programs that access the database. They must develop accessing strategies that best suit the needs of their applications and take full advantage of the database management system. VAX DBMS provides interactive DBQ to assist programmers in this task.

Initially, DBQ helps programmers become acquainted with the DML environment through interactive sessions; programmers use DBQ to test program logic by retrieving, updating, and storing database records interactively. This capability shortens the development time for database programs by allowing programmers to create sound accessing strategies before incorporating these routines into a program. In addition, DBQ can show you how to "navigate" a network database by displaying diagrams of records and their relationships.

When the routines work correctly and are to be incorporated into a program, DBMS creates User Work Areas (UWAs) for your program. UWAs contain the data definitions needed to access the database.

UWAs are automatically created for database programs written in COBOL, FORTRAN, BASIC, BLISS, C, DIBOL, PASCAL, and PL/I. For all programs that have UWAs, subschema definitions are automatically extracted from the CDD and copied into the program at compile time.

4.2.3 Database Performance and Tuning

VAX DBMS enhances system performance with a two-fold approach:

- It supplies parameters you can set for optimum performance. These parameters control the management of your system's routine operations.
- It provides monitoring and performance analysis tools that let you tune your database for best performance.

You can use these monitoring tools to locate performance bottlenecks and change parameters (for example, the size of buffers or the granularity of locks) to improve performance.

VAX DBMS also increases performance through the use of a run-time copy of your database data definitions. These definitions are stored in a database root file. The database root file provides the Database Control System exclusive, immediate access to all definitions pertinent to your database. This prevents conflicts with other uses of the data definitions stored in the CDD.

The following features help provide optimal performance of database operations:

- Space area management

  The use of space area management pages (SPAMs) improves database free-space search performance, especially useful when a database is nearly full.

- Space allocation

  Space allocation lets the DBA specify the exact storage space required for a database data item. Optionally, the DBA can let the system allocate space as needed, with the system compressing data items where possible. The system also compresses database key (dbkey) pointers.

  Other space allocation features are:

  - The ability to tune the system to use fragmented records properly. The use of fragmented records means that the system need not maintain a strict one-to-one correspondence between the size of database pages and the size of the largest database record.
  - The implementation of sorted sets, which provides prefix and suffix compression for sort keys.
  - The ability to separate records into area files, to specify which records should be stored near each other, and to spread area files across disk volumes.

- Buffer allocation

  The buffer allocation management operations let the DBA specify the appropriate number and lengths of buffers to provide maximum data flow within the database system.

You can monitor database performance with two DBO functions:

- The DBO/ANALYZE command

  This command displays statistics for database area files, space usage for the pages in each area requested, and space usage for records and sets. The DBO/ANALYZE command lets a DBA see how various buffer quantities and lengths affect disk-read statistics.

- The DBO/SHOW commands

  These commands (DBO/SHOW STATISTICS, DBO/SHOW SYSTEM, and DBO/SHOW USERS) let you study the processing characteristics of your database. These commands display information about the number of transactions attempted, verb successes and failures, database reads and writes, database monitor activity, and active users. They can help determine the locking characteristics of a multi-user application.

DBO also provides commands to set and adjust database parameters.

4.2.4 Security Features

Several levels of security control are available:

- Normal VMS file protection protects the database files themselves.
- CDD security features protect data definitions in the CDD.
- The security schema protects database objects (data items, records, and sets).
- The DBO utility provides two security control commands. DBO/PERMIT USER maps the security schema information to the User Identification Codes (UICs) you provide. DBO/GRANT COMMAND lets you control which users are allowed to issue a given DBO command.

VAX DBMS uses standard VMS file security to protect database storage area files. However, VAX DBMS restricts unprivileged users from accessing sensitive data, even through such VMS utilities as DUMP.

Note that although a data definition language is a part of the CODASYL model, the security schema definition is a significant enhancement to VAX DBMS DDL that provides security functions well beyond the scope of the CODASYL model.

4.2.5 DECnet Network Access

VAX DBMS is fully supported by VAX DECnet, allowing access to databases on remote nodes. You can bind to a database on another node from a DML program, with callable DBQ, or from interactive DBQ.

4.2.6 Operation in a VAXcluster

VAX DBMS in a VAXcluster environment allows concurrent, multiple-processor database access. VAX DBMS automatically recovers your database if a processor in your VAXcluster fails, and it provides optional after-image journaling to further protect the integrity of your VAXcluster database.

In a properly configured VAXcluster environment, VAX DBMS can give you virtually uninterrupted access to your database. The distributed lock manager provides cluster-wide synchronization of resources. VAX DBMS uses the distributed lock manager to synchronize cluster-wide access to the database root file, to trigger the automatic recovery process when a node fails, and to coordinate concurrent access to a database from processes running on different nodes.

See the *VAX DBMS Database Maintenance and Performance Guide* for more information about using VAX DBMS in a VAXcluster environment.

4.3 Product Summary

The following sections summarize the features of VAX DBMS by listing and briefly describing the major software components:

- Data Definition Language (DDL)
- Database Control System (DBCS)
- Data Manipulation Language (DML)
- Database Query (DBQ) Utility
- Database Operator (DBO) Utility
- HELP Facilities
- The Installation Verification Procedure (IVP)
- Demonstration (DEMO)

4.3.1 Data Definition Language (DDL)

DDL lets you create four types of definitions:

- The schema is a logical definition of the records, sets, and areas that make up a database. The schema definition is the only type of definition you must write (subschema, storage schema, and security schema definitions can be produced automatically by DDL).
- The storage schema definition describes the physical characteristics of database records, sets, and areas as they are stored. A storage schema also defines the placement of records within the database, the representation of data items within a storage record, and storage set parameters. Most database tuning is accomplished by modifying the storage schema.
- The subschema definition describes a logical subset of the database in terms of records, sets, and realms (a collection of one or more areas). Subschemas provide different views of the database to allow for different user needs, special requirements of application programming languages, and limited security.
- The security schema definition describes the access rights to all database objects. DML statement access rights can be defined for areas, records, items, and sets.

The DDL compiler checks your definitions for errors in syntax. If your definition is error-free, DDL stores it in the VAX CDD. You can also use DDL compiler commands to modify your definitions.

4.3.2 Database Control System (DBCS)

The DBCS controls the operation of VAX DBMS at runtime. The DBCS is a shareable image that is linked with any application program that accesses the database. It provides full, concurrent access capabilities (storage, retrieval, update, and deletion) to database records on behalf of user programs, monitors database usage, and acts as an intermediary between VAX DBMS and the VMS operating system.

The DBCS ensures the integrity of the data in your databases by automatically locking records that have been modified, records represented by currency indicators, and records on keeplists. The DBCS uses the locking services of the VMS lock manager.

4.3.3 Data Manipulation Language (DML)

DML lets the database programmer retrieve, update, and store records using CODASYL-compliant commands. The specific database operations supported are CONNECT, DISCONNECT, ERASE, FETCH, FIND, GET, MODIFY, RECONNECT, and STORE.

Boolean expressions with EQ, NE, LE, LT, GT, GE, AND, OR, NOT, MATCHES, and CONTAINS operators can be used in the FIND and FETCH statements.

FREE and KEEP operations save and delete your database context.

COMMIT, READY, and ROLLBACK operations control the permanence of all database operations.

ALSO, EMPTY, MEMBER, OWNER, TENANT, NULL, and WITHIN conditions test the state of the database, currencies, and keeplists.

DML provides the following eight READY mode options that allow a programmer to specify processing intentions:

CONCURRENT RETRIEVAL	CONCURRENT UPDATE
PROTECTED RETRIEVAL	PROTECTED UPDATE
EXCLUSIVE RETRIEVAL	EXCLUSIVE UPDATE
BATCH RETRIEVAL	BATCH UPDATE

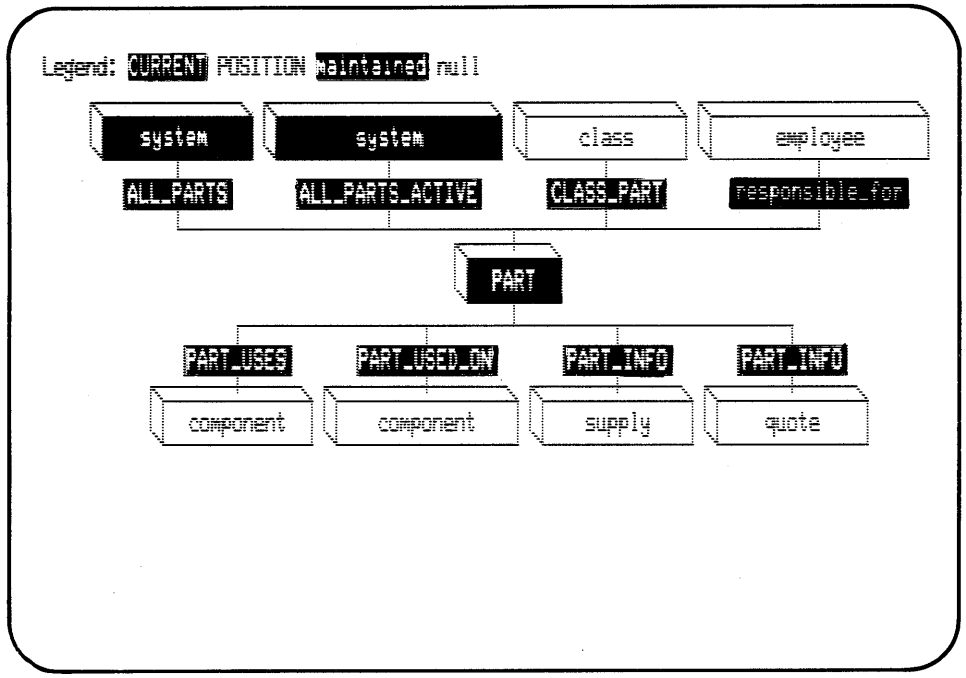
In addition, the DML BIND operation can map your process to a local database, a remote database, or to a mixture of such databases at the same time.

4.3.4 Database Query (DBQ) Utility

DBQ has an interactive and a callable mode of operation. Interactive DBQ provides online, procedural database access capabilities by processing DML statements interactively. It also provides DISPLAY, IF-THEN-ELSE, INITIALIZE, LOOP, MACRO, MOVE, PRINT, SET, SHIFT, and SHOW statements.

When used on a VT100, VT125, or VT200 terminal, interactive DBQ can optionally generate split-screen terminal displays. The bottom half of the screen shows the commands entered and a textual response. The top half of the screen graphically illustrates database access paths with a currency diagram similar to a Bachman diagram. As you navigate through the database, the current position in a subschema is displayed. This feature is an excellent learning tool for introducing database concepts to new users.

VT100- and VT200-compatible terminals show the currency diagram two-dimensionally. VT125-compatible terminals show the diagram three-dimensionally. Figure 4-1 shows a currency diagram as it would appear on a VT125 terminal.



MK-01309-00

Figure 4-1: Currency Diagram on a VT125

4.3.5 Database Operator (DBO) Utility

The DBO utility performs a wide range of database functions:

- Creating, initializing, and deleting databases
- Loading and unloading databases
- Controlling access to DBO commands and to the database through security schemas
- Reporting on VAX DBMS information stored in the CDD
- Extracting and deleting DDL information from the CDD
- Verifying the integrity of internal database structures on line in CONCURRENT, PROTECTED, and EXCLUSIVE modes
- Producing formatted database dumps for an after-image journal (AIJ), and recovery journal (RUJ)
- Producing full and incremental database backup off line
- Restoring the database from backup and journal files
- Controlling and displaying the status of the DBCS monitor process
- Creating a statistical analysis of database space usage
- Displaying DBMS statistics for active databases
- Generating a user work area (UWA) for application programs

4.3.6 Help Facilities

VAX DBMS provides extensive help facilities for the interactive Database Query (DBQ) utility, the Database Operator (DBO) utility, the DBALTER facility, and the DDL compiler. The help files contain all necessary information on the use of each of these facilities. They also contain complete specifications for writing DML statements and schema, subschema, storage schema, and security schema data definitions.

4.3.7 The Installation Verification Procedure (IVP)

The IVP verifies the correct installation of all VAX DBMS components and builds the PARTS database that is used in examples throughout the VAX DBMS documentation. The IVP also verifies whether the correct hardware/microcode needed to run VAX DBMS is installed.

4.3.8 Demonstration (DEMO)

VAX DBMS provides DEMO, an online, self-paced demonstration package that uses the PARTS database. DEMO provides a quick means to become familiar with database creation and manipulation. This demonstration is designed to help you get started developing your own database using VAX DBMS. It is divided into eight modules. They are:

Database Definition
Schema Definition and Compilation
Creating the Database Using the Defaults
Loading a Database
Database Query Language Retrieval
Optimizing the Database (Creating the Database Using Options)
Database Query Language Update
Using Database Manipulation Language
Securing a Database
Database Utilities

These modules are designed to follow each other but after working through them you may want to view them selectively.

You invoke DEMO by first setting your default VMS directory to the directory to contain your sample database. You then set your CDD default (CDD$DEFAULT) to a CDD node to which you have write access. You then type:

```
@SYS$COMMON:[SYSTEST.DBM]DBMDEMO
```

4.4 Types of VAX DBMS Product Kits

VAX DBMS provides two types of kit:

- VAX DBMS

  This is the full development kit. It lets you create DBMS databases and the application programs that access the databases

- VAX DBMS RUN-TIME ONLY

  A run-time only version of VAX DBMS is available after the purchase of a fully supported VAX DBMS license. The run-time only version of VAX DBMS provides all features and facilities except the DDL compiler, the FORTRAN/DML preprocessor, and the precompiler.

  The purpose of the run-time only version is to support the execution of applications on one machine that have been developed on other machines using the application development version of the product.

VAX TDMS 5

5.1 Overview of VAX TDMS

A forms product controls input and output to and from the terminal. VAX TDMS is a forms product that lets you use forms to collect and display information on a terminal. It offers a wide range of features that make forms management easy and thus let you realize significant savings in developing and maintaining forms applications. TDMS is a fully integrated member of the VAX Information Architecture, supported by the VMS operating system.

VAX TDMS offers two major advantages over traditional systems:

- TDMS reduces development and maintenance costs by using record and form definitions that exist outside the application program.
- TDMS provides device independence by letting you write the program without concern for the device on which the application runs.

5.2 Programmer Productivity

VAX TDMS significantly reduces the programming costs associated with developing and maintaining an application by letting you replace portions of the application program with definitions that are created and stored outside the application program. These definitions include:

- Form definitions, which define data input requirements and the appearance of the form
- Record definitions, which define the data type, structure, order, and length of the records used by the application
- Request definitions, which define the exchange of information between the program and the terminal

Because these definitions are not part of the program, it is sometimes possible to change the application without changing the application program itself.

For example, suppose you develop a personnel application that uses forms. Six months after the application is up and running, the personnel department tells you that employee identification numbers will change from five-digit to six-digit numbers. If the application has been developed without TDMS, you incur the major cost of revising, recompiling, and debugging program code. With TDMS, however, the process is greatly simplified: because the definitions that must be changed are outside the program, you may not need to change any program code.

TDMS definitions are described more fully in the following sections.

When you use TDMS, your application program does not control I/O to and from the form. The application program's primary functions are to:

- Call and execute requests
- Provide access to the database used by the application
- Handle run-time errors that might otherwise cause corruption of data

The application programmer need not be concerned with mapping data between forms and records, since this is done entirely by the request. In many applications, TDMS can reduce the number of programming statements and error messages from the application program, using requests to undertake these functions.

As a result, the program in a TDMS application can be viewed as a generic algorithm that executes a series of requests (or routines) and reads information from and/or writes information to a database.

5.3 Device Independence

With VAX TDMS, the application program does not concern itself with the device that the terminal operator uses. Terminal manipulation (such as cursor control, scrolling, and video highlighting) is defined by the form and by the request. Thus, terminal manipulation is wholly independent of the application program.

VAX TDMS applications and utilities can be run on the following terminals:

VT100 series
VT200 series (in VT100 mode)
DECmate
Professional

5.4 Elements of a TDMS Application

Every TDMS application includes the following elements:

- An application program
- One or more record definitions
- One or more form definitions
- One or more requests
- One or more request library definitions
- One or more request library files

The following sections describe these elements in more detail.

5.4.1 The Application Program

In a TDMS application, the program:

- Reads data from and writes data to the database (VAX DBMS, VAX Rdb/VMS, or VAX RMS files)
- Uses the TDMS programming calls to:
  - Open and close request library files
  - Open and close I/O paths to the terminals
  - Execute requests
  - Read text from or write text to the reserved message line on the terminal
  - Cancel a call in progress
  - Signal the return status for TDMS call errors
  - Activate a facility (Trace) that traces the action of a request
- Provides for error processing

In short, the program identifies the requests that are to be executed. The request controls the flow of data between the record and the terminal, and the program controls the flow of data between the record and the database.

A TDMS application can be written in any VAX language that conforms to the VAX Procedure Calling and Condition Handling standard. In addition, application programs written in VAX COBOL, VAX BASIC, VAX FORTRAN, VAX PASCAL, VAX DIBOL, VAX C, VAX RPGII or VAX PL/I can benefit from TDMS's ability to extract record definitions from the CDD, thus eliminating the need to redefine records.

5.4.2 Record Definitions

A record definition identifies the data type, structure, and length of the records used in an application. For application programs that are written in languages that support the CDD, you need only refer to the record definition that already exists in the CDD rather than redefine the record in your program.

You create record definitions with either the VAX CDD Data Definition Language (CDDL), VAX DATATRIEVE, VAX DBMS, or VAX Rdb/VMS. Record definitions are always stored in the CDD.

5.4.3 Form Definitions

A form definition lets you specify the appearance of the terminal at run time. You can control background text, cursor position, scrolled areas, help texts, and video characteristics. You can also use form definitions to restrict the data that the operator is allowed to enter.

The **form definition** describes the screen image displayed on the terminal when the application executes. The form definition contains the information that identifies:

- The screen image of the form. The screen image includes the location of fields and background text as well as video highlighting. **Background text** is text that is always displayed when the form is displayed; **fields** are locations on the form where data can be collected or displayed.
- The location, length, and picture type of each field.
- A set of attributes for each field on the form. These attributes apply certain conditions or characteristics to fields. For example, **field attributes** can require that an operator complete a field in its entirety or return only uppercase data to the record. **Field validators** are special field attributes that you can assign to any field on a form, requiring that the terminal operator's input be within a specified range, that it match an item from a specified list, or that it conform to the requirements of a check-digit algorithm.
- The location of scrolled regions. **Scrolled regions** let the operator enter or see

many lines of data on a few lines of the form. TDMS lets you use vertically scrolled regions on a form for input or output fields.

- The name of a help form, which the terminal operator can display at run time.

The VAX TDMS Form Definition Utility (FDU) checks the syntax of form definitions and stores them in the CDD if it finds no errors.

5.4.4 Request Definitions

A TDMS request is the key element in a TDMS application; it controls the information that is displayed on the terminal and collected from the operator.

As the result of instructions specified by a request, TDMS can:

- Display a form
- Allow data to be entered on the form and transferred to the record (where a program can retrieve and process it)
- Allow data to be transferred from the record (where it was stored by a program) and displayed on the form
- Conditionally perform the above operations based on a value previously entered by the operator or returned by the program
- Allow the operator to use predefined keys that can return "constant" data to the record

TDMS requests are created by the VAX TDMS Request Definition Utility (RDU), which also stores the requests in the CDD.

TDMS permits data that is to be collected or displayed on a single form to be sent to or from any number of records. More complex requests provide additional capabilities, such as the inclusion of conditional instructions.

Note that TDMS also performs data type conversion during the execution of a request, converting from text to the data types of receiving record fields on input and converting the data types of record fields to text format on output.

TDMS does not require any special structure for records, nor does TDMS require exact matches between record and form definitions. TDMS lets you map any combination of records or parts of records to a form or part of a form. The only requirement is that the data type and length of the mapped fields be consistent with the mapping rules of TDMS. Thus, you do not have to restructure your existing record definitions in order to use them in a TDMS application.

5.4.5 Request Library Definitions

A **request library definition** is a list of one or more requests. To use a request in a TDMS application, you must name the request in at least one request library definition.

The request library definition is created by the Request Definition Utility, and it is stored in the CDD. You can use any number of request library definitions in an application.

The request library is contained in a **request library file**, an RMS file that has a default file type of .RLB. The RLB file includes the binary representations of:

- Any requests named in the request library definition
- Any form definitions identified in any request
- Any record information identified in any request

You use the Request Definition Utility (RDU) to build an RLB file. If you modify a request, form definition, or record definition after an RLB file has been built, you must rebuild the RLB file in order to incorporate the changes into the application at run time.

At run time, the program can execute a request only if the request is in an RLB file and the RLB file has been opened by the program (with calls to TDMS routines). When the program executes a request, the request, record information, and form definition are retrieved from the RLB file, not from the CDD.

5.5 TDMS Utility Programs and the Trace Facility

VAX TDMS provides two utility programs to create, store, and modify definitions: the TDMS Form Definition Utility (FDU) and the TDMS Request Definition Utility (RDU).

In addition, TDMS provides a Trace facility that lets you monitor the action of a TDMS application program at run time and thus aids in debugging.

5.5.1 The TDMS Form Definition Utility

The TDMS Form Definition Utility (FDU) lets you create or modify form definitions and store them in the CDD. Using the form editor that is included in FDU,

you can:

- Create a screen image of the form, including:
  - Background text
  - Fields
  - Video highlights that can be activated whenever the form is displayed and/or when a field is available for operator input
  - The screen background (dark or light) and number of columns (80 or 132)
- Assign specific attributes, validation procedures, and input order to fields

5.5.2 The TDMS Request Definition Utility

The TDMS Request Definition Utility (RDU) provides all of the capabilities you need to:

- Define and modify requests and store them in the CDD
- Define and modify request library definitions and store them in the CDD
- Build request library files

RDU includes a validation procedure to ensure that:

- The syntax used in each request is valid
- Each form definition named in the request exists in the CDD
- Each record definition named in the request exists in the CDD
- The data mappings between form and record definitions are consistent with TDMS data mapping rules

RDU also creates, validates, and stores request library definitions and verifies that each request named in a request library definition exists in the CDD.

RDU is also used to build request library files (RLB files). To build the RLB file, RDU retrieves the requests named in the request library definition and the form and record information identified in each request. RDU then creates the RLB file. When building the RLB file, RDU validates each request to make sure that form and record definitions exist, that the request syntax adheres to RDU syntax rules, and that all input and output mappings are legal. Of course, RDU lets you

turn off this validation mode if you are writing the request before you have written the form or record definitions. However, RDU always validates request library files and does not build an RLB file if it detects serious errors.

5.5.3 The Trace Facility

VAX TDMS provides the Trace facility to let you monitor the action of a TDMS program at run time. This facility describes the actions taken during the execution of requests and TDMS calls. Trace is a particularly useful tool when debugging programs that use conditional requests.

5.6 Types of VAX TDMS Kits

VAX TDMS provides two kits:

- VAX TDMS

  VAX TDMS is the full development kit and contains all the components needed to write, test, and compile TDMS applications. This includes the RDU and FDU utilities.

- VAX TDMS RUN-TIME

  VAX TDMS RUN-TIME is the run-time kit for VAX TDMS and provides run-time support for existing TDMS applications. To run a TDMS application on a system with the run-time kit installed, you need only copy the application and its associated request library files onto the system.

VAX ACMS 6

6.1 Overview of VAX ACMS

The VAX Application Control and Management System (ACMS) makes the development, maintenance, and use of online applications easier. Examples of online applications include:

- Operations support, such as order administration, personnel administration, inventory control, and scheduling
- Inquiry and information retrieval, such as examining customer or employee records for reference or for decision support
- Accounting, including accounts payable and receivable, funds transfer, foreign exchange, and payroll

Online applications share certain characteristics. They generally have a moderate to large number of terminal users on the system at the same time. Many of the terminal users have little or no computer experience. The same tasks are available to many of these users, and these tasks do predictable, structured work, such as adding items to inventory, updating a reservation list, or displaying employee records. Moreover, these tasks use the same set of data files or databases.

VAX ACMS was designed to address problems inherent in the development of online applications:

- Complexity

  Online applications often have many users performing many different tasks. Furthermore, these applications tend to grow in the number of functions they perform and in the number of users they serve. VAX ACMS lets you structure your application in an understandable way, thus easing the complexity problem and allowing tasks to be easily added to the application.

- System availability

  Online applications must have high availability lest the business activities they support be stopped or delayed. VAX ACMS provides a way to restart an application more quickly in the event of a system failure. Further, VAX ACMS lets you specify recovery procedures that ensure data integrity; without ACMS, these recovery procedures would have to be coded into each program that accesses data.

- Resource sharing

  Because most online applications are large, they put heavy demands on computer processing power and memory resources. VAX ACMS lets you share system resources among the tasks in an application while avoiding most of the overhead normally associated with process startup, namely the opening and closing of files, the readying of databases, and so on.

The primary principle behind VAX ACMS is the separation of application development from application control and the user interface. In essence, VAX ACMS provides the structural framework with which to build an online application. The following sections discuss this structural framework in more detail.

6.2 Application Development With VAX ACMS

The goal of ACMS is to reduce application development and maintenance costs and increase programmer productivity without sacrificing efficient use of system resources. ACMS does this by providing a way of implementing the tasks in an application that is different from those provided by the VMS operating system or by the other VAX layered products.

In contrast to traditional application development tools, which require detailed knowledge of the system on which the application is implemented, ACMS lets you create task definitions that control what a task does and how it is processed. These definitions, called **multiple-step task definitions**, replace significant parts of program code with simple, direct statements. ACMS provides Application Definition Utility (ADU) clauses for creating task definitions. These and other ADU definitions are stored in the VAX Common Data Dictionary (CDD).

There are two types of steps in a multiple-step task. An **exchange step** handles terminal I/O, usually by means of a VAX TDMS request. The request uses forms for input and output. A **processing step** does the computation or database work needed by the task. It uses a subroutine or procedure written in a programming language such as VAX COBOL or VAX BASIC, a VAX DATATRIEVE command or procedure, a DCL command or procedure, or a VMS image. At the end of each step, you can define one or more **actions** that determine what the task does next.

Figure 6-1 shows the definition for a simple task that writes a new employee record to a file. The task first calls a TDMS request that asks the user for information about the employee. When the information is complete, the task calls a program to write that information to a file. If an error occurs in writing the information, the task returns to the exchange step to display an error message.

```
CREATE TASK ADD_EMPLOYEE
  WORKSPACE IS ADD_EMPLOYEE_WORKSPACE;
  BLOCK WORK
    EXCHANGE
      REQUEST IS GET_EMPLOYEE_INFORMATION
        USING ADD_EMPLOYEE_WORKSPACE;
    PROCESSING
      CALL PERSADD IN PERSONNEL_SERVER
        USING ADD_EMPLOYEE_WORKSPACE;
      ACTION
        CONTROL FIELD IS PERSADD_RETURN_STATUS
          "ERROR"   :  GO TO PREVIOUS EXCHANGE;
          "SUCCESS" :  EXIT TASK;
        END CONTROL FIELD;
  END BLOCK WORK;
END DEFINITION;
```

Figure 6-1: Multiple-Step Task Definition

Multiple-step tasks use workspaces to pass information between steps. In the definition shown in Figure 6-1, the workspace named ADD_EMPLOYEE_WORKSPACE passes information from the exchange step to the processing step.

Tasks developed using VAX ACMS can use VAX DBMS or VAX Rdb/VMS databases, or VAX RMS files. If a task uses VAX DBMS or VAX Rdb/VMS, recovery actions can be controlled by the task definition, further simplifying the development and maintenance of the application. ACMS does not itself provide journaling or recovery facilities.

Structuring the task into exchange (terminal I/O) steps and processing steps makes the task definition easier to understand and maintain. In addition, the separation of terminal I/O from processing lets ACMS dedicate different, specialized VMS processes to each kind of work. ACMS system processes can be used to handle the terminal I/O for many users. Another kind of process, called a server, can be dedicated to computation, database interaction, and other processing work.

Server processes can be used by the processing steps in many tasks without having to be started and stopped for each task. A server process can handle the processing step for one task while other tasks do terminal I/O; the same server process can handle processing for a second task while the first does terminal I/O. This specialization of processes can significantly reduce the resources, including memory, that the application needs.

There are two kinds of servers:

- **DCL servers** handle VMS images, DCL commands, DATATRIEVE commands, DATATRIEVE procedures, and other processing work that can be run from DCL command mode.
- **Procedure servers** handle subroutines written in VAX BASIC, VAX COBOL, or other VAX languages.

Because servers can be used by many tasks, they are defined in a task group definition. The **task group** defines resources that can be shared by many tasks. These resources include TDMS request libraries, VMS message files, ACMS workspaces, and servers.

In addition to the ADU clauses for defining tasks and task groups, VAX ACMS provides a facility to help the application programmer develop ACMS applications. The ACMS Task Debugger lets you debug tasks without setting up an application and its menus. With the Task Debugger, an application programmer can start servers and tasks. While a task is running, the programmer can set breakpoints, examine and change workspace contents, and use the VAX Symbolic Debugger to control processing steps. The commands and qualifiers are similar to those of the VAX Symbolic Debugger.

In summary, the application development features of ACMS let you create an online application using well-defined and easily-understood pieces. Further, ACMS lets you group these pieces so that they work more efficiently with less system overhead.

The following section describes the application control features of ACMS.

6.3 Application Control with VAX ACMS

In addition to supplying the operational environment for VAX ACMS applications, VAX ACMS can also be used to monitor and control existing applications running under the VMS operating system.

The application control features of ACMS address four different types of users:

- **Application managers** set up menus, define applications, control user access to applications and tasks, and monitor and maintain applications.
- **ACMS operators** control and monitor the day-to-day operations of ACMS applications.
- **System managers** authorize users and terminals for access to ACMS, and ACMS applications for access to the VMS operating system.
- **Terminal users** select and run tasks from ACMS menus.

These terms represent roles, not job titles. In many organizations, a single person performs more than one of these roles.

The tools that VAX ACMS provides for these users make it easy to:

- Set up or change menus that let terminal users easily select the tasks they want to run
- Control which users and terminals have access to ACMS
- Control which users can run which tasks in an application
- Control what resources are available to process the tasks in an application
- Control the startup and shutdown of applications
- Monitor application use and performance
- Change ACMS parameters to improve performance

The Application Definition Utility supplies a set of English-like clauses for defining menus and for defining the operational characteristics of ACMS applications. For example, ADU lets you define access control lists that specify which tasks a user is allowed to access in an application. These access control lists can be the same for all tasks in an application or can differ from task to task. Figure 6-2 shows an access control list that makes a Display Parts List task available to a group of users.

```
CREATE APPLICATION INVENTORY_CONTROL
      .
      .
      .
  TASK ATTRIBUTE
    DISPLAY_PARTS_LIST : ACCESS CONTROL LIST
                           ID [100,*] ACCESS EXECUTE;
  END TASK ATTRIBUTE;
END DEFINITION;
```

Figure 6-2: Task Access Control List

Application definitions are stored in the CDD so they can be more easily maintained and controlled.

Terminal users can use ACMS menus to select tasks. Although ACMS provides a standard menu format, the format can be modified to suit the needs of different terminal users. In addition, terminal users can bypass menus and select tasks by typing entry names after the "Selection:" prompt. Certain terminal user commands provided as part of the terminal user interface let users display or bypass

menus. Other terminal user commands let users get help on using ACMS menus, cancel active tasks, select tasks from undisplayed menus or distributed applications, and exit from ACMS.

The separation of menu and form definitions from the procedural code is another example of the structure ACMS imposes on the application. Because of this separation, you need not worry about the type of terminal being used for the application.

6.4 Distributed Applications with VAX ACMS

It is often costly to run an application in a distributed environment. Either the network support must be implemented in application code, or the terminal user must explicitly SET HOST to the node of the network where the application is located. ACMS supports distributed processing. That is, a task developed with ACMS that runs on one node of a network (either VAXcluster, local area network or wide area network) can be selected by terminal users on other nodes. No special programming is required. (The only restriction is that the task do all terminal I/O in exchange steps.) Consequently, the development of distributed applications is much easier with ACMS than with traditional methods.

Thus you might place the terminal I/O portion of an application on one node and the database I/O portion on another. This would let you use a MicroVAX computer as a "terminal server" for an application whose database exists on a VAX 8600 computer.

See the *VAX ACMS Application Management Guide* for more information about creating distributed ACMS applications and for converting existing ACMS applications to a distributed transaction processing environment.

6.5 Additional VAX ACMS Utilities

In addition to supplying a set of terminal user commands and various clauses for defining applications and menus, VAX ACMS also supplies components for managing, using, and running ACMS applications.

The ACMS Operator Facility provides a set of commands for controlling applications. For example, with these commands an ACMS operator can start an application, making its tasks available to users. Or the operator can stop the application so that the tasks are not available and the application does not tie up any system resources. Other ACMS Operator commands display information about applications, tasks, users, and ACMS components.

A second VAX ACMS component, the Audit Trail Facility, helps ACMS operators and application managers monitor the use of ACMS. An Audit Trail logging facility gathers information about task selections, user logins, and other events, and it records this information in a log file. You can then use the Audit Trail Report Utility to format information from the log file into a report. The report

can include all information from the file; it can also be selective, including information about only one user, for example. Similarly, the information gathered by the Audit Trail logging facility can include all applications or can be restricted to one or more applications.

VAX ACMS provides two utilities that control access to ACMS: the User Definition Utility and Device Definition Utility. The User Definition Utility defines which VMS users are authorized to log in to ACMS. It also defines which menu the user sees upon logging in to ACMS or, alternatively, defines the user as one who sees a selection prompt rather than a menu after logging in. The Device Definition Utility defines which terminals can access ACMS and whether those terminals log directly in to ACMS. It also defines which task-submitting agents can submit tasks that run under user names other than their own. With these utilities, users and terminals can be restricted to ACMS, restricted to the VMS operating system, or given access to both. The utilities are similar to the AUTHORIZE Utility, the system management tool provided by the VMS operating system for authorizing VMS users.

The **Application Authorization Utility** allows a system manager to use definitions to authorize applications for installation and to describe allowed characteristics for each application. For example, it describes which users are authorized to install the application using the ACMS/INSTALL Operator command. It also defines which user names the servers in an application can have and the user name under which an application can run.

The sixth major component provided by VAX ACMS is the **ACMSGEN** utility. This utility, similar to the VMS SYSGEN utility, lets system managers change ACMS system parameters, such as how many users can log in, the user names under which ACMS processes run, and the priorities of those processes.

6.6 Types of VAX ACMS Product Kits

VAX ACMS provides three kits:

- VAX ACMS

  (This was formerly known as the full Product Set -- VAX ACMS and VAX ACMS/AD.) VAX ACMS is the full development kit and contains all the components needed to create and control ACMS applications.

- VAX ACMS RUN-TIME

  (This was formerly known as VAX ACMS.) VAX ACMS RUN-TIME is the run-time kit for VAX ACMS. It lets you run applications and change application attributes (for example, menu definitions) but does not allow the creation of new applications.

- VAX ACMS REMOTE

  This kit contains all the ACMS components needed to create a "front-end" environment on one system and access an ACMS application on a remote node.

VAX DATATRIEVE 7

7.1 Overview of VAX DATATRIEVE

VAX DATATRIEVE is a query language and general-purpose data management tool. It is a versatile language with both procedural and nonprocedural characteristics. DATATRIEVE lets you:

- Define data in RMS files
- Store, update, retrieve, and display data from RMS files, VAX DBMS databases, and VAX Rdb/VMS databases
- Query online data
- Write reports
- Display data graphically
- Format screens with either VAX TDMS or VAX FMS
- Access records from files and databases that are distributed on a DECnet network

All these options are available in both an interactive environment (through the DATATRIEVE user interface) and a programming environment (through the DATATRIEVE call interface).

DATATRIEVE can be used by computer professionals and by people with little or no computer experience. DATATRIEVE operates effectively in commercial, technical, scientific, industrial, and educational environments.

The amount of experience you need to perform a task with DATATRIEVE depends on the type of task. People with very little computer experience can quickly learn to use DATATRIEVE to create reports or make *ad hoc* queries.

However, people also use DATATRIEVE to create prototypes of application programs; to use DATATRIEVE as a prototyping tool, you need to know about files and computer applications, that is, more than you would need to know for reports and ad hoc queries.

DATATRIEVE's versatility is also apparent in the data it can access and in the ways it can process that data. For example, you can use DATATRIEVE to query a personnel data file to determine which employees work in a particular department. You can also use the same personnel file to produce a report with a statistical analysis of employee compensation by experience level. And you could still perform either task if the data were to reside in an Rdb/VMS or DBMS database rather than in an RMS file.

DATATRIEVE can also be useful in a distribution facility with an order processing system. In this setting, you could extract sales data by territory and plot the results in pie charts or bar graphs.

In manufacturing, you can use DATATRIEVE to make queries about process control data, with DATATRIEVE's forms capability providing the interface for data entry.

7.2 Comparing DATATRIEVE With Other Computer Languages

DATATRIEVE syntax is more English-like than that of most other computer languages. More importantly, DATATRIEVE has nonprocedural characteristics; thus, you can often simply tell DATATRIEVE what information you want, instead of specifying how to obtain the information.

DATATRIEVE handles many programming language functions automatically, without the need for most of the separate manipulation statements common in programming languages. For instance, DATATRIEVE:

- Finds data files and opens them
- Performs input and output operations
- Formats data for output
- Converts data types automatically
- Handles errors and conditions such as end-of-file

As a result, you can avoid writing many lines of procedural code and thus get applications running quickly. In addition, DATATRIEVE statements can be more readable than the equivalent code in another programming language. For example, a typical programming language might retrieve the records of all employees

named Foster like this:

```
LOOP:

  READ EMPLOYEE-FILE
  AT END EXIT

  IF LAST_NAME NOT = "FOSTER" THEN
    GO TO LOOP
  END IF

  PRINT FIRST_NAME, LAST_NAME, ADDRESS...
  GO TO LOOP
```

In DATATRIEVE, all this code becomes:

```
PRINT EMPLOYEES WITH LAST_NAME = "FOSTER"
```

7.3 Managing Information with DATATRIEVE

DATATRIEVE is a tool for turning data into information. Using DATATRIEVE, you can:

- Store and modify data
- Retrieve data and display it on a terminal, record it in a file, or print it on paper
- Define data in a way that fits your information management needs
- Format data in reports
- Represent data in graphs
- Use VAX TDMS or VAX FMS forms to format the terminal screen for input and display of data
- Get access to data files distributed across a network
- Call any of the information services listed above from a program written in a high-level VAX programming language

The following sections describe these capabilities in more detail.

7.3.1 Defining Data

Creating a DATATRIEVE information management application is a two-phase process. In the first phase, you define the data to be used in the application. You need to define the data only once to establish a foundation on which to build the application. In the second phase, you use DATATRIEVE statements to process

the data associated with these definitions.

The data definition phase of a DATATRIEVE application is usually much simpler than that of applications in other languages because you need only point to existing definitions in the CDD. These existing definitions can be record definitions (for RMS files), schema and subschema definitions (for DBMS databases), or relation and view definitions (for Rdb/VMS databases).

Of course, DATATRIEVE also lets you create record definitions and store them in the CDD. When you create a record definition with DATATRIEVE, you can use the VALID IF clause to specify the values that fields are allowed to contain.

The data definition process involves establishing DATATRIEVE constructs called domains. **Domains** represent relationships between actual physical data and descriptions of data. DATATRIEVE performs all data management in terms of domains.

In its simplest form, a DATATRIEVE domain definition consists of:

- A domain name
- The name of a record definition
- The name of a data file or database

When you define a domain, the domain definition itself is inserted into the CDD, where it can be used by a variety of DATATRIEVE queries and procedures.

The Application Design Tool (ADT) is a DATATRIEVE utility that simplifies the process of defining domains. Operating interactively, ADT presents you with a series of simple questions. Your responses to the questions provide ADT with information to define a domain, define a record, and create a data file.

Domains need not match a record definition exactly; you can create a special kind of domain, called a DATATRIEVE view, which can specify a subset of the fields in a record definition or span several record definitions and data files. Thus you can define a domain that provides information from other domains. A view domain provides a logical view of data stored in one or more files. You can use a view domain just as you use a simple domain.

Here is an example of a simple domain definition:

```
DEFINE DOMAIN PERSONNEL USING PERSONNEL_RECORD ON PERSONNEL.DAT;
```

The record definition PERSONNEL_RECORD describes the data you want to use. The data file PERSONNEL.DAT contains the data. The PERSONNEL domain connects the description with the data.

To use a domain, you must first get access to it with the READY command:

```
READY PERSONNEL
```

After you ready a domain, you can instruct DATATRIEVE to display data with a statement such as:

```
PRINT FIRST 2 PERSONNEL
```

In response to this statement, DATATRIEVE checks the record definition, gets the data requested from the file, and displays the following lines on your terminal:

```
                    FIRST       LAST                START              SUP
 ID      STATUS     NAME        NAME      DEPT      DATE       SALARY  ID

00012 EXPERIENCED CHARLOTTE   SPIVA       TOP   12-Sep-1972 $75,892 00012
00891 EXPERIENCED FRED        HOWL        F11    9-Apr-1976 $59,594 00012
```

If you want to put this information in a file, you can specify an output file:

```
PRINT FIRST 2 PERSONNEL ON FILE.DAT
```

You can also send the information to a line printer:

```
PRINT FIRST 2 PERSONNEL ON LP:
```

DATATRIEVE also has a relational facility for linking domains dynamically. Using the DATATRIEVE CROSS clause, you can join data from separate files in a single statement.

If your system is part of a DECnet network, you can also define domains as remote, so that the record definition and data file can exist on one system and be accessed from another. For example, if your computer is VAX1 and you want access to a domain on computer SYSTM2, you can use this command:

```
READY PERSONNEL AT SYSTM2
```

Now you can display data on your terminal as though the data and record definition were stored on VAX1:

```
PRINT FIRST 2 PERSONNEL
```

While DATATRIEVE can be very simple, it can also be very powerful and versatile. It is possible to construct a single DATATRIEVE view that combines data from an RMS file, a DBMS database, and an Rdb/VMS database.

7.3.2 Storing and Modifying Data

DATATRIEVE lets you, on an ad hoc basis, write data to a file or database or change existing data. You use the DATATRIEVE STORE and MODIFY statements for these purposes.

Before modifying or storing data, DATATRIEVE performs validation checks specified by VALID IF clauses in the DATATRIEVE record definition or by a VERIFY clause in a STORE or MODIFY statement.

To store new records in a domain, you must ready the domain for WRITE access and issue the STORE command. DATATRIEVE then prompts you for the value of each elementary field in the new record. For example:

```
DTR>READY PERSONNEL WRITE
DTR>STORE PERSONNEL
Enter ID: 87422
Enter EMPLOYEE_STATUS: EXPERIENCED
Enter FIRST_NAME: GABBY
Enter LAST_NAME: WEILER
Enter DEPT: T32
Enter START_DATE: 26-AUG-1984
Enter SALARY: 18750
Enter SUP_ID: 87289
DTR>
```

You can modify the data in existing records with equal ease.

7.3.3 Data Retrieval

Data is used to make decisions, generate reports and graphs, and facilitate the operation of an enterprise. DATATRIEVE lets you retrieve stored data with a set of statements. You need not be concerned with the underlying data structure or with the physical location of the data.

DATATRIEVE's data retrieval statements consist of verbs modified by **record selection expressions (RSEs)**. An RSE specifies a subset of the data records in one or more domains. One statement can get the answer to a casual query or produce a detailed report.

A typical data retrieval statement is:

```
FIND PERSONNEL WITH START_DATE AFTER "01-Jan-1982"
```

This statement establishes a collection of records. It might yield a response such as:

```
[50 records found.]
```

Subsequent FIND statements can narrow down this CURRENT collection of 50 records. For example:

```
FIND CURRENT WITH DEPARTMENT EQUAL "SALES" OR "MARKETING" AND
ZIP_CODE EQUAL 02138
```

DATATRIEVE might then respond:

```
[4 records found.]
```

You can then use the PRINT statement to display the information on the terminal screen, record it in a file, or print it on paper. For example, you can display the data on your terminal screen by typing:

```
PRINT ALL NAME, ADDRESS, PHONE
```

If there are complex retrieval statements that you use often, you can define a procedure that includes the statements. Then, each time you need that information, you simply run the procedure instead of typing in the statements.

7.4 Writing Reports With DATATRIEVE

One major reason for organizing and maintaining data is to make the information available to the people who need it. DATATRIEVE's Report Writer helps you present this information in attractive and comprehensive reports.

Managers, secretaries, and other users often need quick access to information about a specific subject. To report this information, you must have rapid access to the data and reliable formatting techniques. With a few simple statements and commands, you can quickly display and accurately summarize data managed by DATATRIEVE.

In addition to query reports, most organizations require detailed summary reports at regular intervals to compare current performance with past performance. These periodic reports are on subjects such as accounts receivable, inventory, and sales. You can use the statistical functions within the Report Writer to summarize the information. Then you can define DATATRIEVE procedures to produce such reports whenever they are needed.

The Report Writer helps you organize your data in a clear, readable format.

It can:

- Center a report name at the top of the page
- Print page numbers and the current date at the upper right of the page
- Set up column headings
- Print a detail line for each record
- Print a summary line for groups of data or for the entire report

You can use the Report Writer's statistical functions to compute values for summary lines. The statistical functions are:

- COUNT
- RUNNING COUNT
- AVERAGE
- TOTAL
- RUNNING TOTAL
- Maximum value (MAX)
- Minimum value (MIN)
- Standard deviation (STD_DEV)

You create a DATATRIEVE report with a series of Report Writer statements called a **report specification**. A report specification begins with a REPORT statement, which specifies the data for the report and the file or device to which DATATRIEVE writes the report. A report specification may contain SET statements, which control the page format and assign heading text; the Report Writer uses built-in defaults for the SET values you do not include. Finally, the report specification ends with an END_REPORT statement.

See the *VAX DATATRIEVE Guide to Writing Reports* for more information.

7.5 Producing Graphics with DATATRIEVE

DATATRIEVE lets you produce three basic types of graphs, or plots, from data in RMS files, VAX DBMS databases, and VAX Rdb/VMS databases. These types of plots are:

- Bar charts
- Line graphs and scattergraphs
- Pie charts

In addition, DATATRIEVE gives you a variety of features that enhance the appearance and usefulness of the three fundamental plot types.

VAX DATATRIEVE graphics can be displayed on any ReGIS device, including:

- VT125, LA100, VT240, and VT241 terminals
- DECwriter IV (LA34-VA), LA12, LA50, and LA100 printers

See the *VAX DATATRIEVE Guide to Using Graphics* for more information.

Index

In this index, a page number followed by a "t" indicates a table reference. A page number followed by an "f" indicates a figure reference.

A

B

C

D

E

F

G

H

I

S

T

U

V

W

VAX Information Architecture
Summary Description
AA-GR93A-TE

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: DISG Documentation ZKO2–2/N53
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, N.H. 03062

Do Not Tear - Fold Here and Tape

Introduction to Application Development with the VAX Information Architecture

---

December 1985

This document gives a step-by-step introduction to developing software applications with components of the VAX Information Architecture.

OPERATING SYSTEM: **VMS**

MicroVMS

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliablity of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

ACMS	DECUS	UNIBUS
CDD	MicroVAX	VAX
DATATRIEVE	MicroVMS	VAXcluster
DEC	PDP	VAX Information Architecture
DECgraph	Rdb/ELN	VMS
DECnet	Rdb/VMS	VT
DECslide	TDMS	digital™

Contents

3 Displaying Data on the Screen

4 Transaction Processing Against a Database

Index

Examples

Figures

Tables

How to Use This Manual

This manual provides an introduction to developing software applications with the products of the VAX Information Architecture.

Intended Audience

This book is intended for application programmers who are new to the VAX Information Architecture. You do not need expertise with the individual products to begin reading this book; however, you should have some familiarity with the VMS operating system, VAX Record Management Services (RMS), and VAX high-level languages. If you do not, you can read:

- The *Introduction to VAX/VMS* for general information about the VMS operating system
- The *Guide to VAX/VMS File Applications* for information about VAX RMS file handling
- The *VAX Software Handbook* for an overview of VAX facilities and capabilities

Operating System Information

To verify which versions of your operating system are compatible with this version of the VAX Information Architecture, check the most recent copy of the following:

- For the VMS operating system -- *VAX/VMS Optional Software Cross Reference Table*, SPD 25.99.xx
- For the MicroVMS operating system -- *MicroVMS Optional Software Cross Reference Table*, SPD 28.99.xx

Structure

There are five chapters and two appendixes in this book:

Chapter 1	Describes the sample applications developed in this manual and introduces the VAX Common Data Dictionary (CDD).
Chapter 2	Explains how to define, create, and work with VAX Rdb/VMS and VAX DBMS databases.
Chapter 3	Discusses how to display data on and retrieve data from the terminal screen in VAX Terminal Data Management System (TDMS) applications.
Chapter 4	Shows how to develop applications with the VAX Application Control and Management System (ACMS) using either a VAX DBMS or a VAX Rdb/VMS database.
Chapter 5	Describes how to use VAX DATATRIEVE software to perform ad hoc queries and generate reports and graphics of either a VAX DBMS or a VAX Rdb/VMS database.
Appendix A	Contains the complete sources for the sample ACMS applications discussed in this manual.

Related Manuals

Before reading this manual, you should read the *VAX Information Architecture Summary Description* for an introduction to the functions and features of the products in the VAX Information Architecture. This applications guide is not an exhaustive discussion of any of these products. Therefore, as you use this manual, you may find it helpful to refer to the documentation sets for the individual products.

Conventions

This section explains the special symbols used in this book:

GOLD-E	The hyphen in key sequences means that you press the first key and then the second key.
Color	Color in examples shows user input.

The software products discussed in this manual are often referred to by simple names. For example, VAX DATATRIEVE software is referred to as DATATRIEVE, and VAX Rdb/VMS software is referred to as Rdb/VMS.

Overview of the VAX Information Architecture 1

The VAX Information Architecture is a set of related products that work together to solve your information management problems. These components include:

- VAX Common Data Dictionary (CDD), a central storage facility for data definitions used by the other components of the architecture and by many VAX high-level languages
- VAX Rdb/VMS, a database management system designed on the relational model
- VAX DBMS, a database management system designed on the network model
- VAX Terminal Data Management System (TDMS), a forms package that manages the display of forms and the movement of data to and from the terminal screen
- VAX Application Control and Management System (ACMS), an application development system for implementing and managing transaction processing applications
- VAX DATATRIEVE, an interactive query language that includes the capability for generating reports and graphics

Each component of the VAX Information Architecture requires definitions of the data on which it operates and instructions for processing the data. By using the CDD as a central repository for data definitions, the VAX Information Architecture components can share definitions and thus work together in complex applications that provide you with maximum flexibility in managing your data.

1.1 Summary of the Sample Applications

This manual describes in detail the development of two transaction processing applications for the AVERTZ Car Rental Company. This company maintains two

separate databases, one for personnel data and another for car rental data. These databases are used in two separate transaction processing systems that permit reservation agents and personnel clerks simultaneously to enter, display, and update the data stored in one of the databases. In both systems, data is displayed on a form on the user's terminal screen, and the user can enter changes directly on the form. Users can also perform interactive ad hoc queries and generate reports and graphics of data collected from the database.

The following chapters describe how the AVERTZ Company used the components of the VAX Information Architecture to implement these applications. All the components used in the sample applications store data definitions in the CDD, which is described in Section 1.2. Appendix A contains the complete sources for the applications, some of which are also used in Chapter 3 to describe the development of a small forms-driven application.

1.2 VAX Common Data Dictionary

The VAX Common Data Dictionary (CDD) provides a central storage location the data definitions used in VAX Information Architecture applications. Without such a repository, you would have to define the data in every piece of an application, leading to redundancy and perhaps inconsistency of data definitions. With the CDD, however, you can include the same definition in every piece of the application that uses the data. If the data changes, you need to change only one definition in the CDD and rebuild the application; the change is automatically reflected.

When you create and compile data definitions with VAX Information Architecture components, they can be stored automatically in the CDD. For example, when you compile a VAX DBMS schema definition, which defines the database records, DBMS inserts it in the CDD. When one definition refers to another (for example, when a TDMS request definition refers to a DBMS record definition), it expects to find the definition it needs in the CDD. Subsequent chapters of this book explain how VAX Information Architecture components interact with the CDD. For more information on using the CDD, see the *VAX Common Data Dictionary User's Guide.*

1.2.1 Structure of the CDD

The CDD is a collection of dictionary objects organized into a hierarchy of dictionary directories. A dictionary object is a definition that belongs to a dictionary directory; for example, DBMS schema definitions and ACMS task definitions are dictionary objects. A dictionary directory simply groups related objects and identifies where they are stored. All directories and objects in a CDD dictionary descend from a top-level directory called CDD$TOP. Figure 1-1 illustrates a possible CDD directory hierarchy.

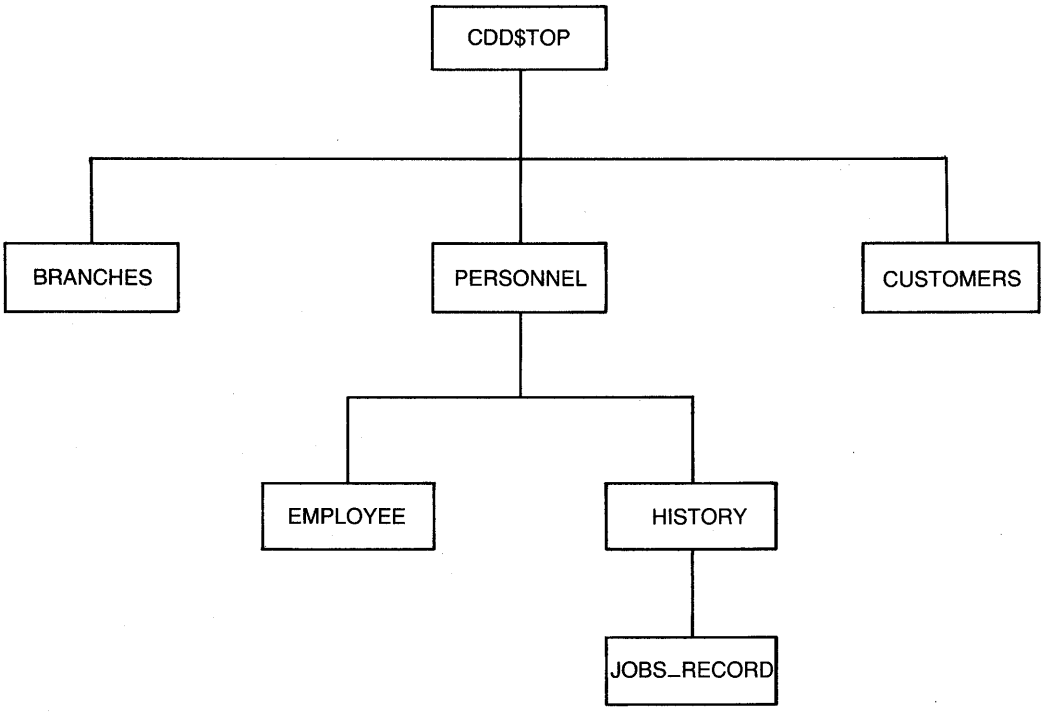


Figure 1-1: Sample CDD Directory Hierarchy

In this figure, BRANCHES, PERSONNEL, and CUSTOMERS are directories under CDD$TOP. Below PERSONNEL are the HISTORY directory and the EMPLOYEE object; below HISTORY is the JOBS_RECORD object. CREDIT_RECORD is an object below CUSTOMERS. There are no objects stored below the BRANCHES directory.

To refer to a definition in the CDD, you must trace the path from CDD$TOP through any intervening directories to the object. You list the name of each directory along the path, separating the names with periods and ending with the name of the object For example, in the hierarchy shown in Figure 1-1, you refer to CREDIT_RECORD as CDD$TOP.CUSTOMERS.CREDIT_RECORD. You can abbreviate references by setting a default CDD directory. If you establish

CDD$TOP.CUSTOMERS as your default directory, you can then omit that part of the path name and refer to the object simply as CREDIT_RECORD. A name that includes no references to the directories that precede an object in the dictionary hierarchy is called the object's **given name**.

Before you define an object that will be stored in the CDD, you need to decide where the definition belongs in the CDD hierarchy. You should create a CDD directory specific to your application and use it to store all the definitions used in the application. The next section shows you how to create directories in the CDD for the personnel and car rental applications.

1.2.2 Creating and Managing Directories in the CDD

You create and manage CDD directories with a utility called the Dictionary Management Utility, or DMU. To enter DMU, you should first define the following symbol at DCL level or in your login command file:

```
$ DMU :== $DMU
```

Then, to invoke DMU, simply type DMU. At the DMU> prompt, you can begin typing DMU commands. You exit from DMU with the EXIT command or CTRL/Z. For more information about DMU commands, type HELP at the DMU> prompt or see the *VAX Common Data Dictionary Utilities Reference Manual.*

The first time you enter DMU, your default CDD directory is CDD$TOP. You use the CREATE command to create a directory below CDD$TOP. To define separate dictionary directories for the personnel and car rental applications, you could create the following directories under CDD$TOP:

```
DMU> CREATE/AUDIT RDBPERS
DMU> CREATE/AUDIT AVERTZ
```

You must have certain privileges to create a CDD directory. To create either of the directories in the preceding example, you must have PASS_THRU and EXTEND privileges at CDD$TOP. If you try to create these directories and receive error messages from DMU, ask your system manager to change your CDD privileges.

Each directory and object in the CDD can have a history list, which contains information about the directory's or object's creation in the CDD and later modifications to the dictionary. You must use the /AUDIT qualifier with the CREATE command if you want to record history list information about a directory, such as the date and time of its creation.

While you are developing an application, you probably want to work most of the time in the directory you created for the application. To establish a directory as your default every time you use DMU, you can define the logical name

CDD$DEFAULT in your login command file. The following example sets CDD$TOP.AVERTZ as your default CDD directory:

```
$ DEFINE CDD$DEFAULT CDD$TOP.AVERTZ
```

You are thus automatically placed in CDD$TOP.AVERTZ when you enter DMU. If you need to move to CDD$TOP.RDBPERS or another directory during a DMU session, you can use the SET DEFAULT command. For example:

```
DMU> SET DEFAULT CDD$TOP.RDBPERS
```

To see the names of the directories and definitions stored in the CDD, you use DMU's LIST command. Suppose you have stored the definition of your car rental database, three task definitions, and a form definition in your default directory, CDD$TOP.AVERTZ. The following command displays the names of the objects stored under that directory:

```
DMU> LIST
   AVERTZSC;1 <DBM$SCHEMA>
   AVERTZ_CHECKIN_FORM;1 <CDD$FORM>
   AVERTZ_CHECKIN_TASK;1 <ACMS$TASK>
   AVERTZ_CHECKOUT_TASK;1 <ACMS$TASK>
   AVERTZ_RESERVE_TASK;1 <ACMS$TASK>
```

DMU also displays the version number of the object and its type (schema definition, form definition, and so forth). As you develop your application and your CDD directory fills up with objects, you can add the /TYPE qualifier to the LIST command to specify which types of objects you want to see. For example, you can list the names of only the task definitions in your directory:

```
DMU> LIST/TYPE=ACMS$TASK
   AVERTZ_CHECKIN_TASK;1 <ACMS$TASK>
   AVERTZ_CHECKOUT_TASK;1 <ACMS$TASK>
   AVERTZ_RESERVE_TASK;1 <ACMS$TASK>
```

When you add the /FULL qualifier to the LIST command and specify the name of an object, DMU shows you the object's complete definition plus information about its creation in the CDD. If you want the definition to be written to a file so that you can print it, you can use the EXTRACT command with the name of the object and an output file specification.

The LIST/FULL command cannot display TDMS form definitions or DBMS and Rdb/VMS definitions of any type. To display a form definition, you must use the TDMS Form Definition Utility (FDU), as described in Chapter 3. Chapter 2 describes how to locate and display a DBMS or an Rdb/VMS definition in the CDD hierarchy that is created for a database.

Setting Up a Database 2

The first step in solving information management problems is to organize the data you need to process. The VAX Information Architecture offers you a choice of two database models for structuring your data: VAX Rdb/VMS, for relational databases, and VAX DBMS, for network databases. The database model you choose depends on several factors, including the amount of data you need to store, the complexity of the relationships among the data, and the frequency of change in the relationships. You must weigh the advantages and disadvantages of each model against the requirements of the data you need to store and of the applications that use the data.

Database implementation consists of two phases, design and definition. In the design phase, you decide which data items you need to store, what the relationships among the data items are, and which database model is appropriate for organizing the data. Once you select a database model, you can proceed to the definition phase, in which you define the necessary constructs for managing your data.

The AVERTZ Company needs to store two kinds of data, personnel and car rental, and wants to keep them in separate databases. Personnel data consists of:

- Personal information, such as an employee's full name, address, date of birth, and employment status (full-time, part-time, or retired)
- Job history, such as the job code, starting and ending dates, department, and supervisor for each job the employee has held
- Salary history, such as the salary amount and the starting and ending dates for the period during which the salary was effective
- Education information, such as colleges attended and degrees awarded

Car rental data consists of:

- Address information about each of the many AVERTZ branch offices
- Information about the types of cars each branch rents
- Customer information
- Reservation information
- Corporate information for companies that have credit accounts with AVERTZ

The AVERTZ Company has a fairly small amount of personnel data with simple relationships but a large amount of car rental data with fairly complex relationships. Thus, a relational database is more suitable for organizing and processing personnel data, while a network database is more suitable for the car rental data. The selection of a database model is not always simple, however; the *VAX Rdb/VMS Guide to Database Design and Definition* and the *VAX DBMS Database Design Guide* can help you decide which model is appropriate for organizing your data.

This chapter explains how to set up the two databases needed for the AVERTZ Company. These databases are used in the rest of the manual to illustrate the other components of the VAX Information Architecture.

2.1 Defining a VAX Rdb/VMS Database

A relational database organizes individual items of data into two-dimensional tables called **relations**. Each row of a table, called a **record**, represents a logical relationship among individual data items, or **fields**. The actual values of the fields distinguish among the many records in a relation. For example, the AVERTZ Company stores personal, job history, salary history, and educational information for each of its employees. The personnel database uses several relations to contain this information; among them is an employee relation whose many records have fields with the same definition but different values. Figure 2-1 shows three records in an employee relation and the actual values for some of the data items in the records.

To define an Rdb/VMS relational database, you must first design the fields and relations you need to organize your data, taking care to normalize your database design. **Normalization** is the process of increasing the flexibility of your database by eliminating redundant information and selecting key fields, as described in the *VAX Rdb/VMS Guide to Database Design and Definition.*

Data Items	ID Number	Last Name	First Name	Middle Initial	City	State
Data Values	00166	Dietrich	Rick		Boscawen	NH
	00169	Gray	Susan	O	Etna	NH
	00174	Myotte	Daniel	V	Bennington	MA

ZK-00022-00

Figure 2-1: Records in the Personnel Database

2.1.1 Designing the AVERTZ Personnel Database

The personnel data can be organized into several relations: one for personal data, one for job history data, one for education data, and so on. Each kind of data can be stored in a distinct relation with a common field for the employee number, allowing you to retrieve the information you need by joining the relations on the basis of matching values in the common field.

Some information, however, is not unique to any given employee; for example, many employees work in the same department and have the same manager. Rather than store complete information about the department in which an employee works, you could store a department code that corresponds to a relation of department information, thus saving storage space in each employee's job history record. One step in normalizing a database is to remove as much redundant information as possible from a relation and store it separately where other relations can refer to it. In the personnel database, you could remove the department information from each job history record and store it in a relation that shares the department code field with the job history relation.

Similarly, many employees may have the same job and employment status, and may have attended the same college. You can remove this redundant information from the individual job history, employee, and education relations and store it in general job, status, and college relations. The employee-specific relations can simply contain codes that correspond to records in the more detailed company-wide relations. Then, with a simple join operation, you can retrieve the information that allows you to interpret the code.

Figure 2-2 illustrates the relationships among the relations in the personnel database. The names in boxes represent the relations in the personnel database. The arrows indicate which records have common fields; the names of the common fields appear without boxes in the figure.

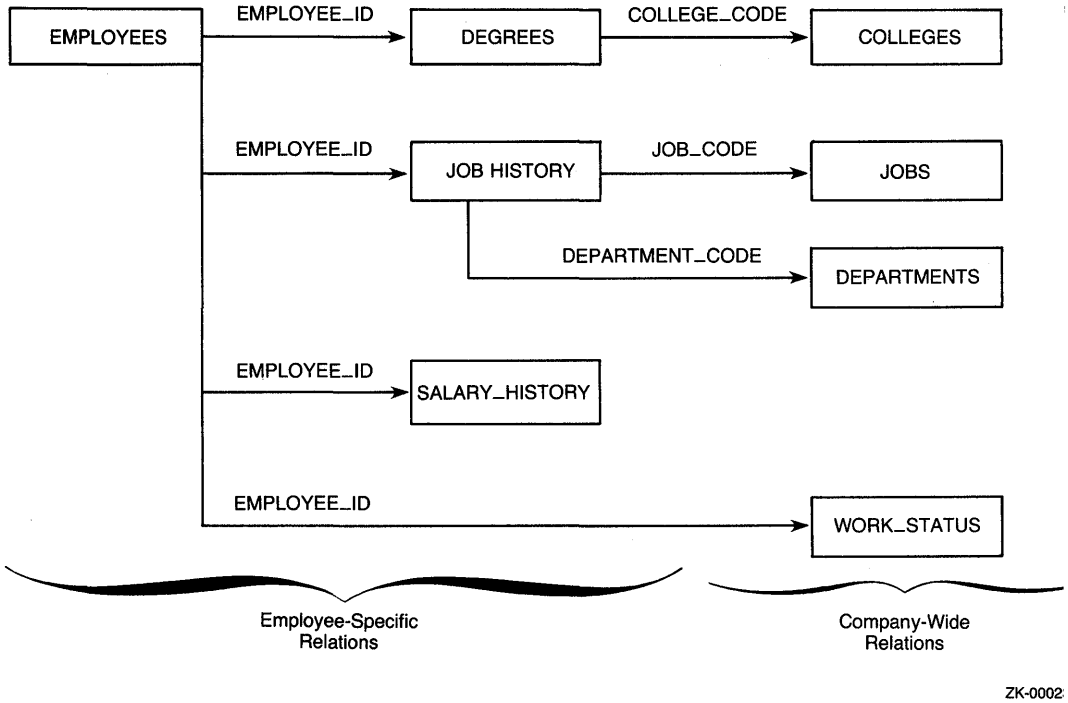


Figure 2-2: Illustration of the Personnel Database

The EMPLOYEES relation shares the EMPLOYEE_ID field with four other relations that hold more specific data about the employee's education, job history, and salary history.

The DEGREES relation uses the COLLEGE_CODE field to indicate the college an employee attended. This field also appears in the COLLEGES relation, which contains the full name and address of various colleges. Similarly, the JOB_HISTORY relation uses the JOB_CODE field to represent an employee's job and the DEPARTMENT_CODE field to represent his or her department. These fields also appear in the JOBS and DEPARTMENTS relations, respectively, along with complete information about a particular job or department. Thus, if you wanted to find out, for example, the manager for the department in which employee 177 works, you would join EMPLOYEES with JOB_HISTORY on the EMPLOYEE_ID field and JOB_HISTORY with DEPARTMENTS on the DEPARTMENT_CODE field.

2.1.2 Defining the Database

Once you are satisfied with the design, you can define your Rdb/VMS database. A database definition is a set of statements for the Relational Database Operator (RDO) utility that define database elements. You can create a text file of these statements with a text editor, such as EDT.

You can define as many as five kinds of elements in a database definition:

- Fields
- Relations
- Views
- Indexes
- Constraints

The following sections describe the definitions of these elements for the AVERTZ Company's personnel database. Section A.1.1 contains the complete database definition.

The *VAX Rdb/VMS Guide to Database Design and Definition* explains in greater detail the subjects covered here, and the *VAX Rdb/VMS Reference Manual* contains reference information on RDO statements. Both of these manuals discuss options that are not illustrated in this manual.

2.1.2.1 Naming the Database -- A database definition creates a database and a directory in the CDD where database definitions can be stored. You use the RDO statement DEFINE DATABASE to name the database and specify its location in the CDD. The following example shows the DEFINE DATABASE statement that creates the personnel database in the CDD$TOP.RDBPERS directory:

```
DEFINE DATABASE 'PERSONNEL' IN 'CDD$TOP.RDBPERS'
```

This statement creates the database file and snapshot files in your default VMS directory. (The snapshot file is used for read-only transactions.) It also creates a database directory under CDD$TOP.RDBPERS. If you do not specify a CDD path name for the database, Rdb/VMS does not create a database directory or store database definitions in the CDD.

2.1.2.2 Defining Fields -- You define each field in your database, using the RDO statement DEFINE FIELD to specify the field's name and data type. An optional DESCRIPTION clause lets you document fields as you define them so that you can keep track of what information each field contains. You enclose your

comments within the delimiters /* and */. You can also supply other information with a field definition, such as:

- **Missing values** that are used if no other value is specified when a relation that uses the field is stored (the MISSING VALUE clause).
- **Validation clauses** that define the allowable values for a field (the VALID IF clause). If you try to store a record with an invalid value in such a field, Rdb/VMS generates an error and does not store the record.
- **Edit strings** that specify how the field is to be displayed by a VAX DATATRIEVE procedure (the EDIT_STRING clause).

The following statements define the fields used in the EMPLOYEES relation:

```
DEFINE FIELD ID_NUMBER
   DESCRIPTION IS /* Generic employee ID */
   DATATYPE IS TEXT SIZE IS 5.

DEFINE FIELD LAST_NAME
   DESCRIPTION IS /* Employee last name */
   DATATYPE IS TEXT SIZE IS 14.

DEFINE FIELD FIRST_NAME
   DESCRIPTION IS /* Employee first name */
   DATATYPE IS TEXT SIZE IS 10.

DEFINE FIELD MIDDLE_INITIAL
   DESCRIPTION IS /* Employee middle initial */
   DATATYPE IS TEXT SIZE IS 1
   EDIT_STRING FOR DATATRIEVE IS 'X.'
   MISSING_VALUE IS ' '.

DEFINE FIELD ADDRESS_DATA_1
   DESCRIPTION IS /* Street name */
   DATATYPE IS TEXT SIZE IS 25
   MISSING_VALUE IS '                         '.

DEFINE FIELD ADDRESS_DATA_2
   DESCRIPTION IS /* Mail stop, apartment number, etc. */
   DATATYPE IS TEXT SIZE IS 25
   MISSING_VALUE IS '                         '.

DEFINE FIELD CITY
   DESCRIPTION IS /* City name */
   DATATYPE IS TEXT SIZE IS 20
   MISSING_VALUE IS '                    '.

DEFINE FIELD STATE
   DESCRIPTION IS /* State abbrevation */
   DATATYPE IS TEXT SIZE IS 2
   MISSING_VALUE IS '  '.

DEFINE FIELD POSTAL_CODE
   DESCRIPTION IS /* Postal code (zip code in US) */
   DATATYPE IS TEXT SIZE IS 9
   MISSING_VALUE IS '         '.
```

```
DEFINE FIELD SEX
   DESCRIPTION IS /* M or F */
   DATATYPE IS TEXT SIZE IS 1
   MISSING_VALUE IS '?'
   VALID IF SEX = 'M' OR SEX = 'F' OR SEX MISSING.

DEFINE FIELD STANDARD_DATE
   DESCRIPTION IS /* Generic date field */
   DATATYPE IS DATE
   MISSING_VALUE IS '17-NOV-1858 00:00:00.00'
   EDIT_STRING FOR DATATRIEVE IS 'DD-MMM-YYYY'.

DEFINE FIELD STATUS_CODE
   DESCRIPTION IS /* A number */
   DATATYPE IS TEXT SIZE IS 1
   MISSING_VALUE IS 'N'
   VALID IF STATUS_CODE = '0' OR
            STATUS_CODE = '1' OR
            STATUS_CODE = '2' OR
            STATUS_CODE MISSING.
```

These statements define global fields that can subsequently appear in any of the relations in the database. The values for all but one of these fields are character strings of the specified lengths; the values for the STANDARD_DATE field are date-and-time stamps.

2.1.2.3 Defining Relations -- A relation definition lists the fields that participate in the relation. After you define all the fields in your database with DEFINE FIELD statements, you can define a relation simply by using the RDO statement DEFINE RELATION to give the relation a name and list the fields it contains.

The following example defines the EMPLOYEES relation:

```
DEFINE RELATION EMPLOYEES.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   LAST_NAME.
   FIRST_NAME.
   MIDDLE_INITIAL.
   ADDRESS_DATA_1.
   ADDRESS_DATA_2.
   CITY.
   STATE.
   POSTAL_CODE.
   SEX.
   BIRTHDAY
      BASED ON STANDARD_DATE.
   STATUS_CODE.
END EMPLOYEES RELATION.
```

Note that two of the fields do not have names listed in previous DEFINE FIELD statements; instead, they use the BASED ON clause to give a new name to a previously defined field. The new name is local to the relation; that is, it can be used only within that relation. Thus, you can define global fields, such as a date field, for common functions and tailor them to specific relations by giving them customized names.

2.1.2.4 Defining Views -- When you design the relations in your database, you normalize your design for efficiency. However, you may frequently want to work with a group of fields that are stored in different relations. For example, you might want to look at an employee's personal and salary information at the same time, combining fields from the EMPLOYEES and SALARY_HISTORY relations. Although your application program could perform a join operation on these two relations every time you want to use them, Rdb/VMS can process the fields more efficiently if you define a view. A view is a "virtual" relation that stores no data; instead, it combines fields from one or more relations in a permanent definition that provides more efficient database access. In your application, you can use a view in the same way that you use a relation for retrieving information, but you cannot store data through a view.

You use the RDO statement DEFINE VIEW to name a view and list the fields it uses. To combine two relations, you must use the CROSS clause and specify the field that the two relations have in common. The following example shows the definition for a view that combines employee information with the current job history record:

```
DEFINE VIEW CURRENT_JOB OF JH IN JOB_HISTORY
   CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
   WITH JH.JOB_END MISSING.
      E.LAST_NAME.
      E.FIRST_NAME.
      E.EMPLOYEE_ID.
      JH.JOB_CODE.
      JH.DEPARTMENT_CODE.
      JH.SUPERVISOR_ID.
      JH.JOB_START.
END VIEW.
```

This view combines the LAST_NAME, FIRST_NAME, and EMPLOYEE_ID fields from the EMPLOYEES relation with the JOB_CODE, DEPARTMENT_CODE, SUPERVISOR_ID, and JOB_START fields from the JOB_HISTORY relation. E and JH are context variables that give temporary names to the relations used in the statement. The CROSS clause specifies that both relations contain the EMPLOYEE_ID field, which allows RDO to locate records in each relation based on the value of that field. Because the value for the JOB_END field is listed as MISSING, this view includes only the JOB_HISTORY records that have no value supplied for the job end date (that is, they represent the employee's current job).

2.1.2.5 Defining Indexes -- An index is a table of field values that Rdb/VMS uses to improve the speed with which it retrieves records from the database. You can define indexes for the fields you use frequently in accessing records. Rdb/VMS then adds an index key to the relation and builds an index using the

specified field or fields. When you perform a database operation that searches for records or joins records based on the indexed field, Rdb/VMS can use the index to locate the records rather than searching sequentially through all the records in a relation.

In deciding which fields in the database need to be indexed, you should choose fields that you use frequently in search and join operations, such as fields that are common to two or more relations. You can also use indexes to prevent a key field from containing duplicate values in two or more records. If you attempt to store a record with a value in the key field that already exists in the database, Rdb/VMS generates an error and does not store the record.

To define an index, you use the RDO statement DEFINE INDEX with a name for the index, the name of the relation to which it applies, and the name of the key field. You can also include the DUPLICATES ARE NOT ALLOWED clause to prohibit duplicate key values. The personnel database defines several indexes for frequently used fields; the following example shows one such index:

```
DEFINE INDEX EMP_EMPLOYEE_ID FOR EMPLOYEES
   DUPLICATES ARE NOT ALLOWED.
      EMPLOYEE_ID.
END EMP_EMPLOYEE_ID INDEX.
```

This example defines the index EMP_EMPLOYEE_ID for the EMPLOYEES relation, using EMPLOYEE_ID as the key field. Because the index definition specifies that duplicate values are not allowed, no two EMPLOYEES records in the database can have the same value in the EMPLOYEE_ID field. The personnel database defines similar indexes for the other relations that contain employee number fields.

2.1.2.6 Defining Constraints -- A constraint is a set of restrictions on the values a field in an Rdb/VMS database can contain. You can place a constraint on a field when you define it, using the VALID IF clause with a DEFINE FIELD statement, as shown in Section 2.1.2.2. Such a constraint, however, can test field values only against constants. A more flexible way of checking the validity of field values is a formal constraint, with which you can test the validity of one field value against the values of other fields in the database.

When you define a formal constraint, Rdb/VMS adds the definition to the database and uses it to check field values that you attempt to store or modify. If the value violates the constraint, Rdb/VMS generates an error message. You can specify whether the constraint should be applied when you update a record (with the STORE or MODIFY statement) or when you commit changes to the database (with the COMMIT statement).

To define a constraint, you use the RDO statement DEFINE CONSTRAINT with a name for the constraint, the name of the field to which it applies, and an expression that describes the constraint. You can also include a CHECK clause to determine when the constraint is evaluated. The personnel database defines several constraints, two of which are shown in the following example:

```
DEFINE CONSTRAINT JH_EMP_ID_EXISTS
   FOR JH IN JOB_HISTORY
   REQUIRE ANY E IN EMPLOYEES WITH
      E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   CHECK ON COMMIT.

DEFINE CONSTRAINT EMPLOYEE_ID_REQUIRED
   FOR E IN EMPLOYEES
   REQUIRE NOT E.EMPLOYEE_ID MISSING.
```

The JH_EMP_ID_EXISTS constraint states that the value of the EMPLOYEE_ID field in the JOB_HISTORY record must exist in the EMPLOYEES relation before the JOB_HISTORY record can be stored. The CHECK ON COMMIT clause specifies that the constraint is not applied until the modified JOB_HISTORY record is committed to the database. The EMPLOYEE_ID_REQUIRED constraint stipulates that a record cannot be stored unless the EMPLOYEE_ID field contains a value.

2.1.3 Creating the Database

When your command file contains all the RDO statements needed to define your database, you can submit the file to RDO and create the database. To use RDO, you should define the following symbol at DCL level or in your login command file:

```
$ RDO :== $RDO
```

Then you can invoke RDO simply by typing RDO at DCL level. RDO responds with the RDO> prompt, and you can begin typing RDO statements or submit an RDO command file. For tutorial information on using RDO, see the *VAX Rdb/VMS Guide to Data Manipulation.*

If RDO finds no errors when it processes your command file, it inserts the database definitions in the CDD directory you specified in the DEFINE DATABASE statement (if you included a CDD path name or given name). In addition, it creates a database file and a snapshot file in your default VMS directory. Rdb/VMS stores definitions and data in the database file. It stores temporary or snapshot versions of database records, used for read-only transactions, in the snapshot file. Therefore, before you execute the command file, make sure your default VMS directory is set to the directory in which you want to store your database.

The following commands create the personnel database defined in the command file PERSDB.RDO:

```
$ SET DEFAULT PERS$EXE
$ RDO
RDO> @PERSDB
```

Because .RDO is the default file type for RDO command files, you need not specify it on the command line. This command creates the PERSONNEL.RDB and PERSONNEL.SNP files in the directory represented by the logical name PERS$EXE. (The default file type for database files is .RDB; the default file type for snapshot files is .SNP.)

When Rdb/VMS stores database definitions in the CDD, it creates a complex structure of field, relation, view, index, and constraint definitions descending from the database directory. When an application processes the data items stored in an Rdb/VMS database, it locates the data items by using a relation definition in the CDD. Because you must specify the CDD path names for the application to use, you should understand how to locate relation definitions in the CDD hierarchy.

Figure 2-3 shows the CDD hierarchy under the PERSONNEL database. When you created the database in the RDBPERS directory below CDD$TOP, Rdb/VMS created a directory named RDB$RELATIONS and stored the relation definitions below it. RDB$RELATIONS is only one of several directories that Rdb/VMS creates in the CDD when you create a database. This figure does not show the other directories but only the path from CDD$TOP to the relation definitions you specify in an application.

As Figure 2-3 shows, the path name to the EMPLOYEES record definition is CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES. You can eliminate the first two directory names if your default CDD directory is always set to CDD$TOP.RDBPERS, but it is safer to include the complete path name in your applications.

If you want to see the definition of an Rdb/VMS field or relation stored in the CDD, you can use SHOW statements in RDO:

- SHOW FIELDS lists all the global fields and their definitions. You can specify a field name to display the definition of a particular field, for example, SHOW FIELD SALARY_AMOUNT. Alternatively, you can add the FOR clause and the name of a relation to see the definitions of all the fields in a particular relation, for example, SHOW FIELDS FOR JOB_HISTORY.
- SHOW RELATIONS lists all the relations in the database and notes whether any of them are, in fact, views.

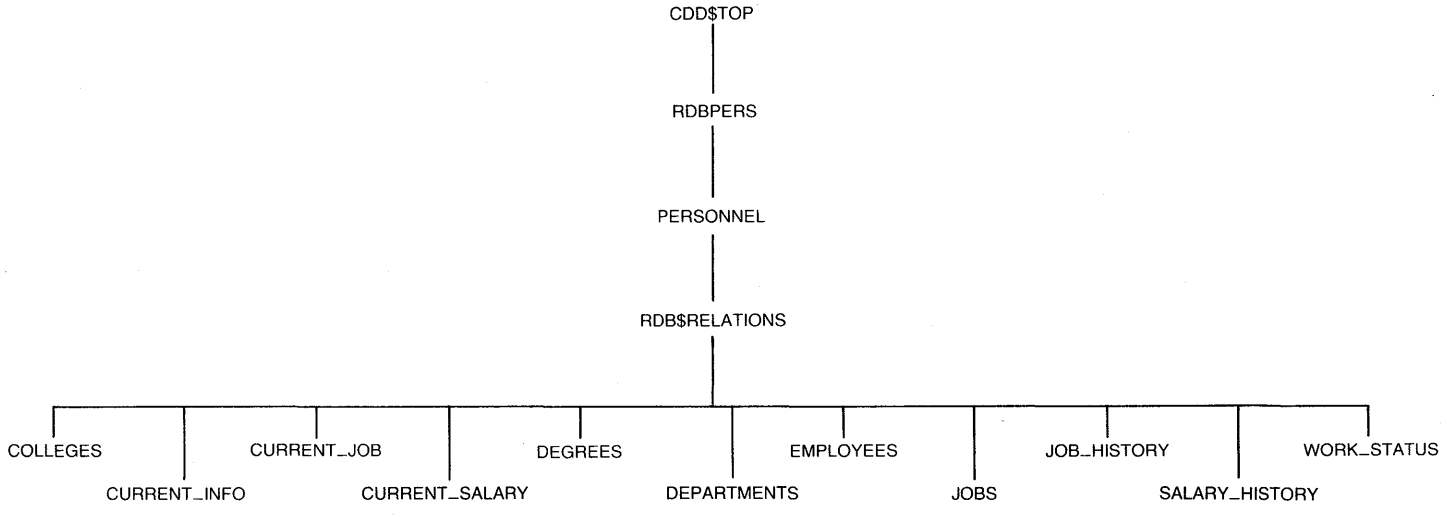


Figure 2-3: CDD Hierarchy of Rdb/VMS Definitions

When you have successfully defined and created your database, you are ready to store data in it. You can use one of several methods to store your data; for example, you can use the STORE statement in RDO, a high-level language program, or a VAX DATATRIEVE procedure. A high-level language program is the recommended method for loading records from VAX RMS files into an Rdb/VMS database. The *VAX Rdb/VMS Guide to Database Administration and Maintenance* discusses load programs in detail.

2.1.4 Working with an Rdb/VMS Database

An application program processes data in an Rdb/VMS database by including statements from the data manipulation language (DML). Before you actually write a high-level language program to use with an Rdb/VMS database, you should test and debug the logic of the program by using RDO in its interactive form. With RDO you can determine whether your DML statements store, retrieve, modify, and delete the appropriate records in the database. When you are sure that your logic is correct, you can incorporate the DML statements into a high-level language program. This section uses RDO examples to illustrate how you might manipulate the data in the personnel database to perform some common transactions.

Rdb/VMS programming is discussed in the *VAX Rdb/VMS Guide to Programming*; reference information about RDO can be found in the *VAX Rdb/VMS Reference Manual*.

2.1.4.1 Accessing the Database -- To work with an Rdb/VMS database, you must first invoke the database with the RDO statement INVOKE, indicating the file specification of the database (.RDB) file. For example:

```
RDO> INVOKE DATABASE FILENAME PERSONNEL
```

This statement invokes the personnel database in your default VMS directory, using the file name PERSONNEL.RDB.

When you are finished working with a database, you end access to it with the FINISH statement:

```
RDO> FINISH
```

If the last transaction has not been completed, FINISH automatically completes it by making permanent any changes made to the database in the transaction. After you issue the FINISH statement, you can invoke another database and continue working with RDO, or you can exit from RDO with the EXIT command or CTRL/Z.

2.1.4.2 Database Transactions -- A set of DML statements that you want to perform together is called a **transaction**. In a transaction, the statements must either execute as a unit or not at all. If only some of the statements execute and others fail, the data in your database could become inconsistent.

You mark the beginning of a transaction with a START_TRANSACTION statement. You also indicate the type of operation you intend to perform by specifying one of the following options:

- READ_ONLY -- You can retrieve but not update records.
- READ_WRITE -- You can both retrieve and update records.

You can also determine the extent to which other users can access the database by reserving relations for certain types of access. If you are performing a READ_ONLY transaction, you can reserve relations for SHARED READ access. This access mode allows other users to read and update records from these relations, but they cannot read the records you are using until your transaction is finished. If you are performing a READ_WRITE transaction, you can reserve relations for the following access modes:

- SHARED READ
- SHARED WRITE -- While you are updating records in a relation, other users can also read and update other records in the relation; however, any changes you make are not available to other users until your transaction is finished.
- PROTECTED READ -- While you are reading records in a relation, other users can read the relation, but they cannot update it until your transaction is finished.
- PROTECTED WRITE -- While you are updating records in a relation, other users can read the relation, but they cannot update it until your transaction is finished.
- EXCLUSIVE READ -- While you are reading records in a relation, other users can neither read nor update the relation until your transaction is finished.
- EXCLUSIVE WRITE -- While you are updating records in a relation, other users can neither read nor update the relation until your transaction is finished.

For example, if you are going to update the EMPLOYEES relation but want to allow other users to update the relation at the same time, you use the following statement:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont> EMPLOYEES FOR SHARED WRITE
```

If you want to reserve all the relations in a database for the same type of access, you can omit the RESERVING clause that names specific relations. For example:

```
RDO> START_TRANSACTION READ_WRITE
```

This statement allows you to access all relations in the database for retrieval and update. Rdb/VMS reserves individual relations within the database as they are used by DML statements in your program.

When your transaction is complete, you can either save any changes you made to the database or cancel them. If do not want to make permanent any changes you made since the last START_TRANSACTION statement, use the ROLLBACK statement:

```
RDO> ROLLBACK
```

If you are satisfied with your changes, use the COMMIT statement to write them to the database:

```
RDO> COMMIT
```

After you end a transaction, the START_TRANSACTION statement that was in effect is canceled. You must enter another START_TRANSACTION statement to begin another transaction.

2.1.4.3 Manipulating Data Within the Database -- The common operations you perform on the data stored in a database are to add, retrieve, modify, and delete records. The examples in this section show the DML statements that perform such operations on the personnel database.

Suppose a new employee is hired and assigned ID number 43517. To store all the necessary information for this employee, you must add a new record to each of four relations: EMPLOYEES, JOB_HISTORY, SALARY_HISTORY, and DEGREES. For example:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont> EMPLOYEES, JOB_HISTORY, SALARY_HISTORY,
cont> DEGREES FOR SHARED WRITE
RDO>
RDO> STORE E IN EMPLOYEES USING
cont> E.EMPLOYEE_ID = '43517';
cont> E.LAST_NAME = 'Marks';
cont> E.FIRST_NAME = 'Gregory',
cont> E.MIDDLE_INITIAL = 'A';
cont> E.ADDRESS_DATA_1 = '309 Park Drive';
cont> E.TOWN = 'Denver';
cont> E.STATE = 'CO';
cont> E.POSTAL_CODE = '80335';
cont> E.SEX = 'M';
cont> E.BIRTHDAY = '15-FEB-1958';
cont> E.STATUS_CODE = '1'
cont> END_STORE
RDO>
```

```
RDO> STORE J IN JOB_HISTORY USING
cont> J.EMPLOYEE_ID = '43517';
cont> J.JOB_CODE = 'SPGM';
cont> J.JOB_START = '6-APR-1983';
cont> J.DEPARTMENT_CODE = 'ENG';
cont> J.SUPERVISOR_ID = '00435'
cont> END_STORE
RDO>
RDO> STORE S IN SALARY_HISTORY USING
cont> S.EMPLOYEE_ID = '43517';
cont> S.SALARY_AMOUNT = '36500';
cont> S.SALARY_START = '6-APR-1983'
cont> END_STORE
RDO>
RDO> STORE D IN DEGREES USING
cont> D.EMPLOYEE_ID = '43517';
cont> D.COLLEGE_CODE = 'HVDU';
cont> D.YEAR_GIVEN = 1979;
cont> D.DEGREE = 'BA';
cont> D.DEGREE_FIELD = 'Arts'
cont> END_STORE
RDO> COMMIT
```

To verify that the records were actually stored, you can retrieve them from the relations, based on the employee ID number. Because RDO displays record fields horizontally on your terminal screen, the display occupies more than 80 characters. Therefore, you can specify only a few fields of each record to make the display more readable. For example:

```
RDO> START_TRANSACTION READ_ONLY
RDO> FOR E IN EMPLOYEES
cont> CROSS J IN JOB_HISTORY
cont> CROSS S IN SALARY_HISTORY
cont> CROSS D IN DEGREES
cont> WITH E.EMPLOYEE_ID = '43517'
cont> AND E.EMPLOYEE_ID = J.EMPLOYEE_ID
cont> AND E.EMPLOYEE_ID = S.EMPLOYEE_ID
cont> AND E.EMPLOYEE_ID = D.EMPLOYEE_ID
cont> PRINT E.LAST_NAME, J.JOB_CODE, S.SALARY_AMOUNT, D.COLLEGE_CODE
cont> END_FOR
 Marks              SPGM    36500.00          HVDU
RDO> COMMIT
```

Note that when you are using RDO interactively, you use the PRINT statement to display the records you retrieve. When you embed DML statements in a high-level language program, you must change the PRINT statement to a GET statement to assign the retrieved value to a program variable.

If this employee moves to a new address, you must modify his EMPLOYEE record accordingly:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont> EMPLOYEES FOR SHARED WRITE
```

```
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = '43517'
cont> MODIFY E USING
cont> E.ADDRESS_DATA_1 = '47 Larimer Square';
cont> E.ADDRESS_DATA_2 = 'Apartment D';
cont> E.POSTAL_CODE = '80332'
cont> END_MODIFY
cont> END_FOR
RDO> COMMIT
```

Updating the address requires only one relation in the database. However, if this employee is given a raise and a promotion, you must modify two relations: you must enter a job ending date in the current JOB_HISTORY record and store a new JOB_HISTORY record for the new job code, and you must enter a salary ending date in the current SALARY_HISTORY record and store a new SALARY_HISTORY record for the new salary. For example:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont> JOB_HISTORY, SALARY_HISTORY FOR SHARED WRITE
RDO>
RDO> FOR J IN JOB_HISTORY WITH J.EMPLOYEE_ID = '43517'
cont> MODIFY J USING
cont> J.JOB_END = '14-MAY-1985'
cont> END_MODIFY
cont> END_FOR
RDO>
RDO> STORE J IN JOB_HISTORY USING
cont> J.EMPLOYEE_ID = '43517';
cont> J.JOB_CODE = 'SANL';
cont> J.JOB_START = '14-MAY-1985';
cont> J.DEPARTMENT_CODE = 'ENG';
cont> J.SUPERVISOR_ID = '00435'
cont> END_STORE
RDO>
RDO> FOR S IN SALARY_HISTORY WITH S.EMPLOYEE_ID = '43517'
cont> MODIFY S USING
cont> S.SALARY_END = '14-MAY-1985'
cont> END_MODIFY
cont> END_STORE
RDO>
RDO> STORE S IN SALARY_HISTORY USING
cont> S.EMPLOYEE_ID = '43517';
cont> S.SALARY_AMOUNT = '39700';
cont> S.SALARY_START = '14-MAY-1985'
cont> END_STORE
RDO> COMMIT
```

The *VAX Rdb/VMS Guide to Data Manipulation* explains DML statements in greater detail. The DML examples in this section illustrate the logic that will be used to process personnel transactions in the sample application in Chapter 4. By using RDO to construct a prototype of your application, you can locate and correct logic errors in the early stages of application development.

2.2 Defining a VAX DBMS Database

A VAX DBMS database organizes individual items of data into records that indicate how the data items are related. The actual values of the data items distinguish among many occurrences of records of the same type. For example, the AVERTZ Company needs to store, for each of its customers, the customer's full name, home address, phone number, and driver's license information. The car rental database contains many occurrences of customer records, all with the same record definition but with different values for the data items.

The various types of records in a VAX DBMS database are in turn organized into sets. In each set, one record type is designated as the set's owner and another record type as the set's member. In the car rental database, one record type identifies the company's customers and another contains information about rental car reservations. A set in which the customer record is the owner and the reservation record is the member represents the relationship between an AVERTZ customer and his or her car reservations. Such a set occurs many times in the database, once for each customer on file. Figure 2-4 shows three occurrences of the customer-reservation set in the car rental database.

In Figure 2-4, the first set is owned by the customer Quinn and has one reservation record as its member. The second set, owned by customer Waite, and the third set, owned by customer Taylor, each have two reservation records as members. The relationships between set owners and members are defined in the database and cannot be changed. DBMS stores pointers that represent the relationships between record types and uses these pointers to locate record occurrences.

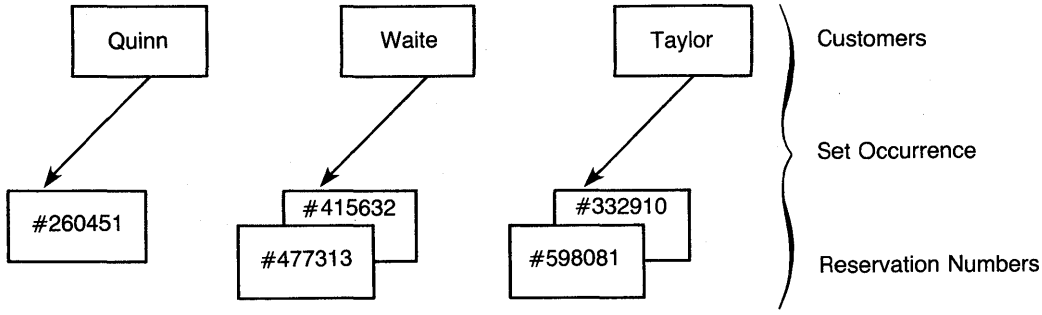


Figure 2-4: Set Occurrences in the Car Rental Database

To set up a DBMS database, you must define the data items, records, and sets in your database and specify the relationships among them. This information forms the schema for the database. Because you might use one database for several different applications, you can define one or more subschemas that specify subsets of the database tailored to the needs of various programs in each application.

2.2.1 Designing the AVERTZ Car Rental Database

Car rental data can be organized into three general groups: customer, location, and company data. For each AVERTZ location, the company needs to store information about the types of cars it rents (compact cars, mid-size cars, and full-size cars) and about the actual cars of each type that it has on hand. For each customer, AVERTZ keeps general information on file and stores a new reservation record when a customer rents a car. When the customer picks up the car, AVERTZ keeps track of which car was assigned so that it knows how many cars have been rented at each location. In addition, some customers work for companies that have credit accounts with AVERTZ, and the validity of these accounts must be checked before a car rental can be charged to the account.

The structure of a DBMS database is frequently represented by a Bachman diagram that shows all the record types in the database and the set types to which they belong. The Bachman diagram for the car rental database is shown in Figure 2-5; the names of the record types are shown in boxes, with set owners connected to set members by an arrow. Names not in boxes are the names of the set types.

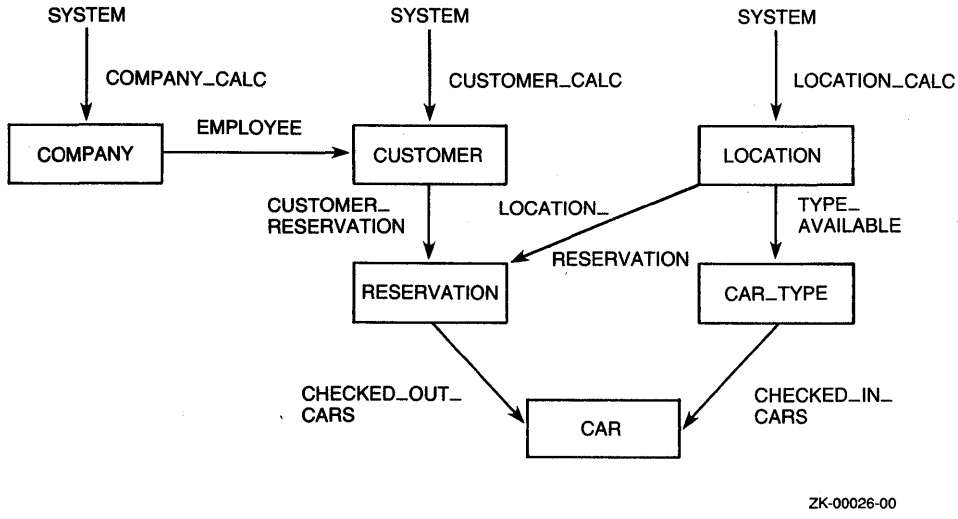


Figure 2-5: Bachman Diagram of the Car Rental Database

The three sets on the top level in Figure 2-5 are called system-owned sets because they represent entry points into the database. SYSTEM is the owner of each set; the member record types are COMPANY in the COMPANY_CALC set, CUSTOMER in the CUSTOMER_CALC set, and LOCATION in the LOCATION_CALC set. These records store information about companies with AVERTZ credit accounts, AVERTZ customers, and AVERTZ branch offices, respectively.

The EMPLOYEE set has COMPANY as its owner and CUSTOMER as its member; it represents the customers who work for companies with AVERTZ accounts. Not all customers necessarily belong to the EMPLOYEE set; customers who rent cars as individuals (that is, who do not charge them to corporate accounts) belong only to the CUSTOMER_CALC set.

The CUSTOMER_RESERVATION set shows the car rental reservations made by a particular customer; CUSTOMER is the owner of this set and RESERVATION is the member. The RESERVATION record is also a member of the LOCATION_RESERVATION set (LOCATION is the owner), which indicates the reservations that have been made for cars at a particular AVERTZ location.

The LOCATION record is the owner of another set, TYPE_AVAILABLE, with CAR_TYPE as the member. This set describes the types of cars (compact, mid-size, and full-size) that can be rented at each branch office. CAR_TYPE in turn is the owner of the CHECKED_IN_CARS set, whose member is the CAR record. The cars of each type that are presently checked in at a location are members of this set.

Finally, a car that has been rented for a specific reservation is represented by the CHECKED_OUT_CARS set, in which the RESERVATION record owns a CAR record.

2.2.2 Defining the Database

Once you are satisfied with the design, you can define the schema for your DBMS database. A **schema definition** is a set of clauses of the DBMS data definition language (DDL) that define database elements. You can create a text file of these clauses with a text editor, such as EDT.

There are four sections in a schema definition:

- The schema entry
- Area entries
- Record entries
- Set entries

The following sections describe the definitions of these elements for the AVERTZ Company's car rental database. Section A.2.1.1 contains the complete schema definition.

The *VAX DBMS Database Design Guide* explains in greater detail the subjects covered here, and the *VAX DBMS Database Administration Reference Manual* contains reference information on DDL clauses. Both of these manuals discuss options that are not illustrated in this manual.

2.2.2.1 Naming the Schema and Areas -- The schema and area entries name the database schema and the areas it uses. An **area** is a subdivision of a database that corresponds to a VAX RMS file that contains the data stored in your database. By restricting records and sets to certain areas of the database, you can sometimes improve database performance by controlling the areas in which records are stored. Section 2.2.4 describes how these areas correspond to VAX RMS files.

This example shows the schema and area entries of the schema definition for the car rental database:

```
SCHEMA NAME IS AVERTZSC

AREA NAME IS COMPANY_AREA

AREA NAME IS CUSTOMER_AREA

AREA NAME IS LOCATION_AREA
```

The schema entry specifies AVERTZSC as the schema name. The area entries define three areas for the car rental database for company, customer, and location data, respectively.

2.2.2.2 Defining Records -- You must include a record entry for every record type in the database. A record entry specifies the name of the record type and the areas in which it can occur. It also lists the individual data items that make up the record type and specifies their data types.

The following record entry defines the COMPANY record type in the car rental database:

```
RECORD NAME IS COMPANY
   WITHIN COMPANY_AREA
      ITEM NAME IS CO_NAME              TYPE IS CHARACTER 25
      ITEM NAME IS CO_ADDR_DATA_1       TYPE IS CHARACTER 25
      ITEM NAME IS CO_ADDR_DATA_2       TYPE IS CHARACTER 25
      ITEM NAME IS CO_CITY              TYPE IS CHARACTER 20
      ITEM NAME IS CO_STATE             TYPE IS CHARACTER 2
      ITEM NAME IS CO_POSTAL_CODE       TYPE IS CHARACTER 9
      ITEM NAME IS CO_PHONE             TYPE IS CHARACTER 10
      ITEM NAME IS CO_CREDIT_CHECK      TYPE IS CHARACTER 2
      ITEM NAME IS CO_DISCOUNT          TYPE IS SIGNED LONGWORD
```

This entry names the record type, COMPANY, and specifies that it occurs within the area of the database called COMPANY_AREA. The COMPANY record has nine data items, or fields; in eight of these fields, the values are character strings of the specified length, while the CO_DISCOUNT field is stored as a signed longword. The SIGNED LONGWORD data type lets you use the field easily in numerical calculations.

2.2.2.3 Defining Sets -- You use a set entry to express relationships between records in your database and to indicate which record type is the owner of a set and which is the member. As the Bachman diagram in Figure 2-5 shows, a record type can participate in more than one set as either the owner or the member; the only restriction is that it cannot be both the owner and the member of the same set. Every set must have an owner record type, but you may want to designate some sets as system-owned. You generally use system-owned sets as entry points into the database.

Besides set ownership and membership, the set entry also describes the insertion, retention, and order of a set's member records. In VAX DBMS:

- The insertion options specify whether a member record is inserted into a set immediately when it is stored in the database (INSERTION IS AUTOMATIC) or inserted only by an explicit CONNECT statement in the application program (INSERTION IS MANUAL).
- The retention options specify whether a record can be removed from a set only if it is being deleted from the database (RETENTION IS FIXED), cannot be removed but can be reconnected to another set occurrence (RETENTION IS MANDATORY), or can be removed without being deleted (RETENTION IS OPTIONAL).
- The order options specify whether a new record occurrence is inserted at the beginning of a set (ORDER IS FIRST), at the end of a set (ORDER IS LAST), immediately after the current record (ORDER IS NEXT), or immediately before the current record (ORDER IS PRIOR).

The car rental database defines three system-owned sets:

```
SET NAME IS COMPANY_CALC
   OWNER IS SYSTEM
   MEMBER IS COMPANY
      INSERTION IS AUTOMATIC
      RETENTION IS FIXED

SET NAME IS CUSTOMER_CALC
   OWNER IS SYSTEM
   MEMBER IS CUSTOMER
      INSERTION IS AUTOMATIC
      RETENTION IS FIXED

SET NAME IS LOCATION_CALC
   OWNER IS SYSTEM
   MEMBER IS LOCATION
      INSERTION IS AUTOMATIC
      RETENTION IS FIXED
```

System-owned sets, such as those shown in this example, are usually defined with FIXED retention because they can occur only once in the database. The ability to reconnect a record to another set occurrence, provided by MANDATORY retention, is useless because there are no other occurrences of a system-owned set. All but one of the sets in the database have OPTIONAL retention, allowing a record to be disconnected from a set occurrence. CUSTOMER_RESERVATION is defined with FIXED retention so that the AVERTZ Company has a record of all the reservations made by a customer.

Most of the sets in the database are defined with AUTOMATIC insertion. The exceptions are the EMPLOYEE set and the CHECKED_OUT_CARS set. They are defined with MANUAL insertion because not every occurrence of the member record types is necessarily a member of these sets. In the EMPLOYEE set, a CUSTOMER record occurrence is a member of the set only if the customer works for a company that has an AVERTZ account. In the CHECKED_OUT_CARS set, a CAR record is a member of the set only if car has been checked out to a current reservation.

For most of the sets, the set entries specify the ORDER IS LAST option; that is, new records are added after all existing records in the set. However, for CUSTOMER_RESERVATION, LOCATION_RESERVATION, and CHECKED_OUT_CARS, the ORDER IS FIRST option declares that when a new reservation record is stored or when a car is checked out by a customer, the record is added in front of all other records in the set. The ORDER options are not used for the system-owned sets because they are stored as CALC sets. CALC sets provide faster record retrieval for sets in which the order of stored records is not important. You cannot specify the ORDER options with CALC sets; they can be used only with CHAIN sets, in which members are accessed sequentially.

The *VAX DBMS Database Design Guide* contains a detailed description of these concepts and can help you decide which choices are appropriate for your database.

2.2.3 Compiling the Schema

When your schema definition is complete, you can use the DDL compiler to compile the schema and store it in the CDD where an application program can locate the database definitions. The DDL compiler generates:

- A default subschema that is identical to the schema
- A default storage schema to describe how the records and sets should be stored in RMS files
- A default security schema to determine the operations and types of access allowed for areas, sets, and records

These additional schemas are also stored in the CDD.

You compile your schema with the DDL/COMPILE command. If the DDL compiler finds no syntax errors in your schema definition, it inserts the schema in your default CDD directory. It also inserts the default subschema, storage schema, and security schema unless you specify otherwise. It determines your default CDD directory by translating the logical name CDD$DEFAULT. Therefore, make sure that you have defined CDD$DEFAULT as the CDD directory in which you want to store your database definitions.

The following DDL/COMPILE command compiles the schema stored in the source file AVERTZSC.DDL:

```
$ DDL/COMPILE AVERTZSC
```

Because .DDL is the default file type for DDL source files, you need not specify it on the command line. This command creates the database definitions in the default CDD directory CDD$TOP.AVERTZ. If you discover after creating a database schema that you need to modify it, you can edit the source (.DDL) file and recompile the schema, using the DDL/REPLACE command.

2.2.3.1 CDD Hierarchy for a DBMS Database -- When the DDL compiler inserts a schema in the CDD, it creates a complex structure of schema, area, set, and record definitions descending from the schema directory. When an application processes the data items stored in a DBMS database, it locates them by using record definitions within a specified subschema definition in the CDD. Because you must specify the CDD path names for the application to use, you should understand how to locate the record definitions within the subschema in the CDD hierarchy.

Figure 2-6 shows the CDD hierarchy for the default subschema under the AVERTZSC schema. When you created the database in the AVERTZ directory below CDD$TOP, the DDL compiler created a directory named DBM$SUBSCHEMAS and stored the subschema definitions below it. DBM$SUBSCHEMAS is only one of several directories that the compiler creates in the CDD when you create a database. This figure does not show the other directories but only the path from CDD$TOP to the record definitions you specify in an application.

As Figure 2-6 shows, the path name to the RESERVATION record definition is CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION. You can eliminate the first two directory names if your default CDD directory is always set to CDD$TOP.AVERTZ, but it is safer to include the complete path name in your applications.

If you want to see the definition of a DBMS record stored in the CDD, you can use the SHOW command in DBQ and specify the record name. You can also list the records, sets, and realms in your database with the SHOW RECORDS, SHOW SETS, and SHOW REALMS commands.

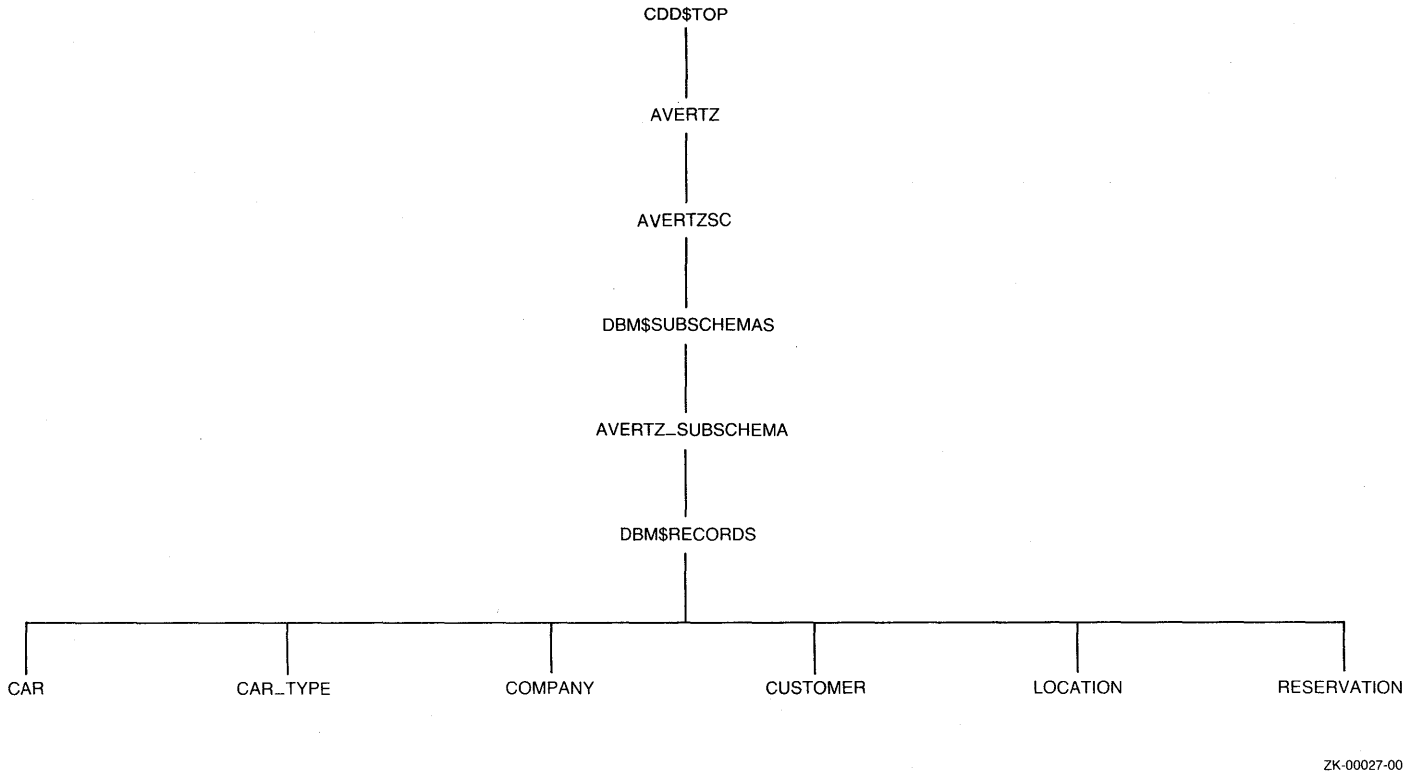


Figure 2-6: CDD Hierarchy of DBMS Definitions

2.2.3.2 Modifying the Default Schemas -- The default subschema, storage schema, and security schema are meant to serve as templates that you can tailor to suit the needs of a particular application. You can create additional subschemas if you want to change certain characteristics of database records or if you want to make only a small subset of the database available to an application. You can modify the storage schema to improve database performance and the security schema to protect the database against unauthorized access. The *VAX DBMS Database Design Guide* and the *VAX DBMS Database Security Guide* can help you decide whether you need to change the default schemas.

In the subschema for the car rental database, it would be useful to define three group fields: one for the customer's full name, one for the reservation number at a location, and one for a customer's reservation number. The first group lets an application program locate a customer by referring simply to the group name rather than to all three fields. Using group fields for the location code and reservation number lets an application program form the customer reservation number from a combination of the location code and a number that is automatically incremented every time a reservation is made at that location. Group fields can be defined only in a subschema.

To edit the default subschema and schemas, you must first extract them from the CDD, using the Database Operator utility (DBO). The following command extracts the default subschema for the AVERTZSC schema:

```
$ DBO/EXTRACT/SUBSCHEMAS/OUTPUT=AVERTZSS AVERTZSC
```

This command creates in your default VMS directory a source file named AVERTZSS.DDL that contains the default subschema. Section A.2.1.2 shows the edited subschema. When you edit a default subschema, remember to change its name on the first line of the definition. If you named the new subschema AVERTZSS, the first line of the subschema definition would be:

```
SUBSCHEMA NAME IS AVERTZSS FOR AVERTZSC SCHEMA
```

The following command compiles the new subschema and inserts it into the CDD:

```
$ DDL/COMPILE AVERTZSS
```

For the AVERTZ car rental database, it is also useful to modify the default storage schema and define the fields CU_LAST_NAME, CU_FIRST_NAME, and CU_INITIAL as the hash keys for the CUSTOMER_CALC set. By doing so, you eliminate the possibility that duplicate records can be stored for the same customer. To edit the default storage schema for the AVERTZSC schema, you first extract it from the CDD as follows:

```
$ DBO/EXTRACT/STORAGE_SCHEMAS/OUTPUT=AVERTZST AVERTZSC
```

This command creates in your default VMS directory a source file named AVERTZST.DDL that contains the default storage schema. Section A.2.1.3 shows the edited storage schema. When you edit the default storage schema, remember to change its name; this example uses AVERTZST for the name of the new storage schema.

The following command compiles the new storage schema and inserts it in the CDD:

```
$ DDL/COMPILE AVERTZST
```

2.2.4 Creating the Database

To create a database from your DDL definitions, you use the DBO/CREATE command, which creates at least two types of files:

- A database root file
- Database area files, one for each area defined in the schema

The root file (default file type .ROO) contains binary versions of the database definitions that your application can use at run time. The area files (default file type .DBS) contain the actual data stored in your database.

The DBO/CREATE command can also create snapshot files for each area of the database, to use in read-only transactions, and an after-image journal file, to use in recovering a corrupted database. By default, the DBO/CREATE command creates these files using the schema you specify on the command line and the default subschema, storage schema, and security schema stored in the CDD. If you stored your own definitions for these schemas in the CDD, you can use qualifiers on the DBO/CREATE command to change the default behavior. See the *VAX DBMS Introduction to Database Administration* and *VAX DBMS Database Design Guide* for more information.

Before creating the car rental database, you should set your default VMS directory to the directory in which you want to store the root and area files. You can then create the database with the following command:

```
$ SET DEFAULT AVERTZ$APPL
$ DBO/CREATE/SUBSCHEMA=AVERTZSS/STORAGE_SCHEMA=AVERTZST AVERTZSC
```

This command creates a root file named AVERTZSC.ROO in the directory represented by the logical name AVERTZ$APPL, using the CDD definitions AVERTZSS and AVERTZST to define a new subschema and storage schema. In addition, it creates three area files, using the area names given in the schema for the file names in the file specifications. Thus, the files COMPANYAR.DBS, CUSTOMERA.DBS, and LOCATIONA.DBS are also created in the directory represented by the logical name AVERTZ$APPL. (The DBO/CREATE command uses only the first nine characters of the area names, excluding special characters such as underscores.)

When you have successfully defined, compiled, and created your database, you are ready to store test data in it. You can use one of several methods to store your data; for example, you can use DBMS's Load facility, a program written in a high-level language, or VAX DATATRIEVE. The Load facility, invoked by the DBO/LOAD command, is the recommended method for loading records from VAX RMS files into a VAX DBMS database. See the VAX DBMS documentation set for more information on the Load facility.

After extensive testing of your database's performance, you might decide to modify one or more of the various schema definitions and recreate and load a new database. During the testing phase, you can use the DBO/ANALYZE and DBO/SHOW STATISTICS commands to see whether the database records are being stored and retrieved efficiently. In addition, you can poll the database users to confirm that their applications process and manage the data correctly. See the *VAX DBMS Design Guide* and the *VAX DBMS Maintenance and Performance Guide* for more information about performance testing.

2.2.5 Working with a VAX DBMS Database

An application program processes data in a DBMS database by including statements from the data manipulation language (DML). Before you actually write a high-level language program to use with a DBMS database, you should test and debug the logic of the program by using the Database Query utility (DBQ). DBQ is an interactive program you can use at your terminal to determine whether your DML statements retrieve, store, modify, and delete the appropriate records in the database. When you are sure that your logic is correct, you can incorporate the DML statements into a high-level language program. This section uses DBQ examples to illustrate how you might manipulate the data in the car rental database to perform some common transactions.

To use DBQ, you should define the following symbol at DCL level or in your login command file:

```
$ DBQ :== $DBQ
```

Then you can invoke DBQ simply by typing DBQ at DCL level. DBQ responds with the dbq> prompt, and you can begin typing DBQ commands and statements. For tutorial information on using DBQ, see the *VAX DBMS Introduction to Data Manipulation*. DBMS programming is discussed in the *VAX DBMS Programming Guide*, and reference information about DBQ can be found in the *VAX DBMS Programming Reference Manual*.

2.2.5.1 Accessing the Database -- An application accesses a DBMS database by means of a subschema. To work with a DBMS database, you must first bind

the database with the BIND command, indicating the root file and the subschema you want to use. For example:

```
dbq> BIND AVERTZSS FOR AVERTZSC
```

This command binds the subschema named AVERTZSS for the AVERTZSC database. DBQ expects to find the database root file in your default VMS directory because you did not specify another location on the command line.

When you are finished working with a database, you end access to it with the UNBIND command:

```
dbq> UNBIND
```

You must have completed the last transaction by either committing or rolling back the database. You can then bind another database and continue working with DBQ, or you can exit from DBQ with the EXIT command or CTRL/Z.

2.2.5.2 Database Transactions -- A set of DML statements that you want to perform together is called a **transaction**. In a transaction, the statements must either execute as a unit or not at all. If only some of the statements execute and others fail, the data in your database could become inconsistent.

You indicate the beginning of a transaction with a READY statement, which readies realms for your transaction to use. A **realm** is a group of one or more areas in the database. If you do not define realms in a subschema, each database area is considered a separate realm.

When you ready a realm, you must also indicate the extent to which other users may or may not access the realms you are using and the type of operation you intend to perform. You can control other users' capabilities by specifying one of the following options:

- CONCURRENT -- Other users can work with the same realms you are using.
- PROTECTED -- Other users can retrieve records from the realms but cannot update them.
- EXCLUSIVE -- Other users can neither retrieve records from nor update the realms.
- BATCH -- If you are only retrieving from realms, other users can update them. If you are updating realms, other users have no access at all to them.

The options that determine the operations you can perform are:

- RETRIEVAL -- You can read but not write records.
- UPDATE -- You can both read and write records.

For example, if you are going to update the CUSTOMER_AREA realm but want to allow other users to retrieve records from that realm while you are updating it, you use the following statement:

```
dbq> READY CUSTOMER_AREA PROTECTED UPDATE
```

If you want to ready all the realms in the subschema with the same options, you can omit the names of the realms. For example:

```
dbq> READY CONCURRENT RETRIEVAL
```

This statement allows other users to use the database while you are retrieving records.

When your transaction is complete, you can either save any changes you made to the database or cancel them. If you do not want to make permanent any changes you made since the last READY statement, use the ROLLBACK statement:

```
dbq> ROLLBACK
```

If you are satisfied with your changes, use the COMMIT statement to write them to the database:

```
dbq> COMMIT
```

After you end a transaction, the READY statement that was in effect is canceled. You must enter another READY statement to begin another transaction.

2.2.5.3 Manipulating Data Within the Database -- The common operations you perform on the data stored in a database are to add, retrieve, modify, and delete records. The examples in this section show the DML statements that perform such operations on the car rental database.

Suppose an existing customer named Taylor wants to reserve a car at the AVERTZ location in Fort Collins, Colorado. To store this reservation, you must first locate the customer record for Taylor and the location record for the Fort Collins branch; then you can insert a new reservation within those occurrences of the CUSTOMER_RESERVATION and LOCATION_RESERVATION sets.

```
dbq> READY PROTECTED UPDATE
dbq> FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING CU_NAME
CU_NAME
  CU_LAST_NAME [CHARACTER (20)] = Taylor
  CU_FIRST_NAME [CHARACTER (15)] = Jennifer
  CU_INITIAL [CHARACTER (1)] = K
```

```
CU_NAME
  CU_LAST_NAME = Taylor
  CU_FIRST_NAME = Jennifer
  CU_INITIAL = K
CU_ADDR_DATA_1 = 264 Palm Drive
CU_ADDR_DATA_2 =
CU_CITY = Indianapolis
CU_STATE = IN
CU_POSTAL_CODE = 46222
CU_PHONE = 3179442090
CU_LICENSE_NO = 464553739
CU_LICENSE_STATE = IN
dbq>
dbq> FETCH FIRST LOCATION WITHIN LOCATION_CALC USING LO_CODE
RESERVATION_ID
  LO_CODE [CHARACTER (2)] = FC
RESERVATION_ID
  LO_CODE = FC
  LO_RES_NUM = 426
LO_NAME = Fort Collins Avertz
LO_ADDR_DATA_1 =
LO_ADDR_DATA_2 = 732 Swift Street
LO_CITY = Fort Collins
LO_STATE = CO
LO_POSTAL_CODE = 80521
LO_PHONE = 3032987654
dbq>
dbq> STORE RESERVATION
RESERVATION_ID
  R_PICKUP_LOCATION [CHARACTER (2)] = FC
  RESERVATION_NUM [CHARACTER (9)] = 306725993
R_CAR_TYPE_CODE [SIGNED LONGWORD] = 2
R_PICKUP_DATE [CHARACTER (6)] = 25-MAY-1985
dbq> COMMIT
```

Because both sets were defined in the schema to have automatic insertion, the RESERVATION record is automatically stored in both the CUSTOMER_RESERVATION and LOCATION_RESERVATION sets. To verify that the record was actually stored, you can try to retrieve it from the LOCATION_RESERVATION set. First you must locate the correct occurrence of the set by finding the location record for the Fort Collins location. Then, because the schema requires a new record to be inserted in this set before all existing records, the new reservation should be the first record in the set.

```
dbq> READY CONCURRENT RETRIEVAL
dbq> FETCH FIRST LOCATION WITHIN LOCATION_CALC USING LO_CODE
RESERVATION_ID
  LO_CODE [CHARACTER (2)] = FC
RESERVATION_ID
  LO_CODE = FC
  LO_RES_NUM = 427
LO_NAME = Fort Collins Avertz
LO_ADDR_DATA_1 = 732 Swift Street
LO_ADDR_DATA_2 =
LO_CITY = Fort Collins
LO_STATE = CO
```

```
LO_POSTAL_CODE = 80521
LO_PHONE = 3032987654
dbq>
dbq> FETCH FIRST RESERVATION WITHIN LOCATION_RESERVATION
RESERVATION_ID
  R_PICKUP_LOCATION = FC
  RESERVATION_NUM = 427
R_CAR_TYPE_CODE = 2
R_PICKUP_DATE = 25-MAY-1985
dbq> COMMIT
```

Suppose that Taylor later calls AVERTZ and asks for a bigger car; you must locate the appropriate reservation, based on the pickup date, and change the car type code from 2 to 3.

```
dbq> READY PROTECTED UPDATE
dbq> FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING CU_NAME
CU_NAME
  CU_LAST_NAME [CHARACTER (20)] = Taylor
  CU_FIRST_NAME [CHARACTER (15)] = Jennifer
  CU_INITIAL [CHARACTER (1)] = K
CU_NAME
  CU_LAST_NAME = Taylor
  CU_FIRST_NAME = Jennifer
  CU_INITIAL = K
CU_ADDR_DATA_1 =
CU_ADDR_DATA_2 = 264 Palm Drive
CU_CITY = Indianapolis
CU_STATE = IN
CU_POSTAL_CODE = 46222
CU_PHONE = 3179442090
CU_LICENSE_NO = 464553739
CU_LICENSE_STATE = IN
dbq>
dbq> FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION USING -
dbq>_R_PICKUP_DATE
R_PICKUP_DATE [CHARACTER (6)] = 052585
RESERVATION_ID
  R_PICKUP_LOCATION = FC
  RESERVATION_ID = 427
R_CAR_TYPE_CODE = 2
R_PICKUP_DATE = 25-MAY-1985
dbq>
dbq> MODIFY R_CAR_TYPE_CODE
R_CAR_TYPE_CODE [SIGNED LONGWORD] = 3
dbq> COMMIT
```

When Taylor arrives to pick up her car on the appointed date, you must find her reservation, see what type of car she asked for, and assign her a specific car to rent. Because CAR_TYPE records are owned by LOCATION records, once you find her reservation, you must find the owner of that reservation within the LOCATION_RESERVATION set; you then find the requested car type at that location and finally a car of that type. To assign a car to the reservation, you disconnect the CAR record from the CHECKED_IN_CARS set and connect it to the CHECKED_OUT_CARS set owned by the current reservation.

```
dbq> READY PROTECTED UPDATE
dbq> FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING CU_NAME
CU_NAME
  CU_LAST_NAME [CHARACTER (20)] = Taylor
  CU_FIRST_NAME [CHARACTER (15)] = Jennifer
  CU_INITIAL [CHARACTER (1)] = K
CU_NAME
  CU_LAST_NAME = Taylor
  CU_FIRST_NAME = Jennifer
  CU_INITIAL = K
CU_ADDR_DATA_1 =
CU_ADDR_DATA_2 = 264 Palm Drive
CU_CITY = Indianapolis
CU_STATE = IN
CU_POSTAL_CODE = 46222
CU_PHONE = 3179442090
CU_LICENSE_NO = 464553739
CU_LICENSE_STATE = IN
dbq>
dbq> FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION USING -
dbq>_R_PICKUP_DATE
R_PICKUP_DATE [CHARACTER (6)] = 052585
RESERVATION_ID
  R_PICKUP_LOCATION = FC
  RESERVATION_NUM = 427
R_CAR_TYPE_CODE = 3
R_PICKUP_DATE = 25-MAY-1985
dbq>
dbq> FIND OWNER WITHIN LOCATION_RESERVATION
dbq> FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE USING CAR_TYPE_CODE
CAR_TYPE_CODE [SIGNED LONGWORD] = 3
CAR_TYPE_CODE = 3
DAILY_RATE_LT_7_DAYS = 30
DAILY_RATE_GT_7_LT_30_DAYS = 25
DAILY_RATE_GT_30_DAYS = 20
DAILY_RATE_FUTURE_USE = 0
dbq>
dbq> FETCH FIRST CAR WITHIN CHECKED_IN_CARS
CAR_NUM = 14858329
CAR_TYPE_CODE = 3
CAR_MAKE = Ford
CAR_YEAR = 85
LICENSE_NUM = 50031380
LICENSE_STATE = CO
dbq>
dbq> DISCONNECT FROM CHECKED_IN_CARS
dbq> CONNECT TO CHECKED_OUT_CARS
dbq> COMMIT
```

The *VAX DBMS Introduction to Data Manipulation* explains DML statements in greater detail. The DML examples in this section illustrate the logic that will be used to process car rental transactions in the sample application in Chapter 4. By using DBQ to construct a prototype of your application, you can locate and correct logic errors in the early stages of application development.

Displaying Data on the Screen 3

Forms-driven applications receive input from and return output to a form that is displayed on a terminal screen. A terminal user supplies the input by typing values at the keyboard to fill in the fields on the form. The output requested by the user is retrieved from a file or database and returned to form fields. Forms-driven applications developed with the VAX Information Architecture components use VAX TDMS to control input received from a terminal screen and output sent back to the screen.

VAX TDMS handles screen input and output by means of forms, requests, and workspaces. A **form** is a screen layout that contains background text and fields for values that are either filled in by the user or displayed by the application. A **request** is a list of instructions for displaying the form on the screen and moving data to a temporary buffer called a workspace. The **workspace** includes fields for all the data being input on or output to a form. An application program written in a high-level language calls requests to display forms, retrieves information that a request stores in a workspace, and uses that information to process records in a master file or database. (Although you can use VAX RMS files to store the data for a TDMS application, this manual does not discuss that option; it assumes that your data is stored in either a VAX DBMS or VAX Rdb/VMS database.) You store form, request, and workspace definitions in the CDD, where an application program can locate them.

To implement a forms-driven application with TDMS, you write a high-level language program that includes data manipulation statements for your database and TDMS programming calls to handle input and output. This chapter shows the development of a VAX COBOL program that uses TDMS programming calls to call requests, display forms, and interact with an Rdb/VMS database. If you were writing a COBOL program that accesses a DBMS database, there is little difference in how you would define forms, requests, and workspaces and write TDMS programming calls. From the TDMS perspective, the only elements specific to a database are the CDD path names to the workspaces you use in the requests.

3.1 A TDMS Personnel Application

One of the many administrative tasks that the AVERTZ Company performs regularly is to record employees' promotions and salary increases in the personnel database. This task combines two kinds of operations:

- An inquiry operation, in which the user supplies an employee number and is then shown the employee's job history and salary history information
- An update operation, in which the user can change the relevant employee data

When the program retrieves the current job history and salary history records from the database, it must keep track of the employee's present job code and salary. After the user updates the information displayed on the form, the program can compare the job code and salary on the form and determine whether the employee received a promotion, a raise, or both. If the employee received a promotion, the program must modify the current job history record to include an ending date for the current job and store a new record for the new job. Similarly, if the employee received a raise, the program must add a salary ending date to the current salary history record and store a new record for the new salary.

The personnel program handles these operations in the following series of steps:

1. Opens the request library file, which contains binary versions of the requests, and opens a channel to the terminal for output
2. Calls a request that displays a form on which the user can type an employee number
3. Uses embedded DML statements to retrieve the current job history and salary history records
4. Calls another request to display the form again, showing the employee number, current department code, job code, starting date for that job, supervisor's employee number, and present salary
5. Uses more DML statements to modify the existing job history and salary history records and to store new records with the updated information
6. Closes the request library and the channel

The following sections describe how to define the forms, requests, workspaces, and request library used in the update program. Section 3.7 shows the complete COBOL program that updates the personnel database.

3.2 Defining Forms

You define a form with the Form Definition Utility, or FDU. To create a form definition, you use FDU's CREATE FORM command and specify a CDD path name or given name for the form. If the form already exists and you want to change some of its characteristics, use the MODIFY FORM command.

To enter FDU, first define FDU as a global symbol at DCL level or in your login command file:

```
$ FDU :== $FDU
```

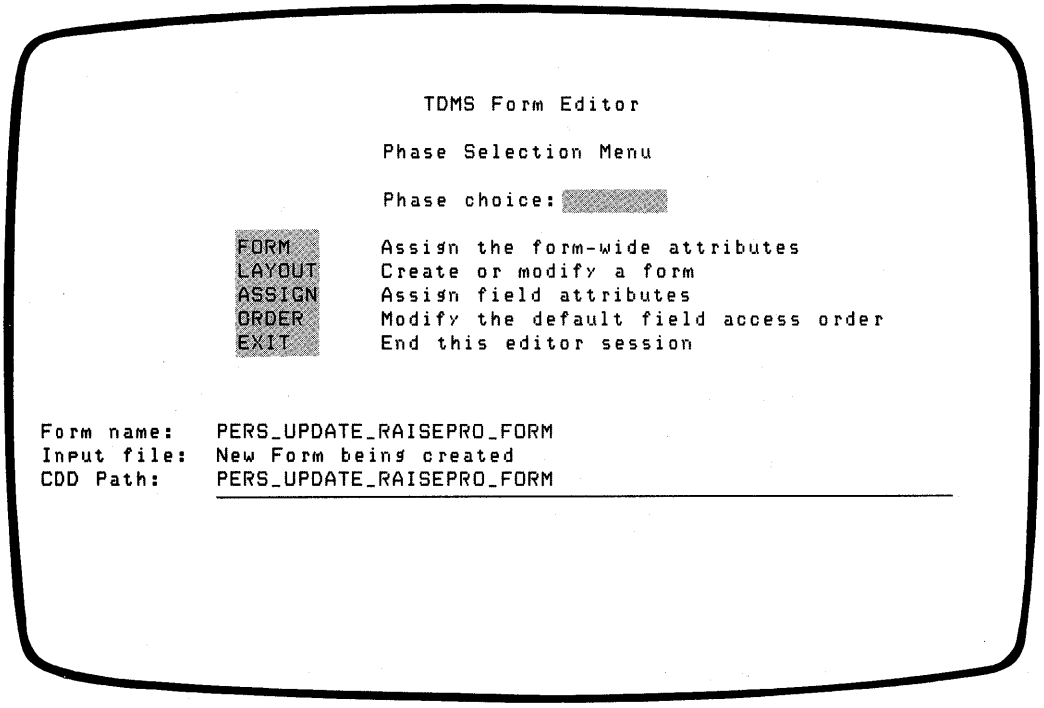
Then, to invoke FDU, simply type FDU. At the FDU> prompt, you can begin typing FDU commands. For detailed information about FDU commands, type HELP at the FDU> prompt or see the *VAX TDMS Forms Manual.*

The personnel program displays a form on which the user types input for both the inquiry and update operations. In the inquiry operation, the user types an employee number that the program can use to retrieve job and salary history information from the database. In the update operation, the user can type new values for the job code, supervisor's employee number, and salary fields. The following command creates the definition for this form:

```
FDU>CREATE FORM PERS_UPDATE_RAISEPRO_FORM
```

Unless you specify a complete CDD path name, the form definition is stored in your default CDD directory; therefore, make sure that your default directory is set correctly before you issue this command (or include the full CDD path name on the command line).

The CREATE FORM command automatically puts you into the Form Editor, with which you define the screen layout of the form. You use the editor to create background text and form fields and to define optional attributes for individual fields. When you enter the form editor, you see the Phase Selection Menu, which asks you to choose one of five editor phases. Figure 3-1 shows the Phase Selection Menu that FDU displays when you create the inquiry/update form.



ZK-00046-00

Figure 3-1: Phases of the TDMS Form Editor

3.2.1 The Form Phase

The Form phase of the form editor lets you assign optional attributes that apply to the entire form, such as the width of the screen, the color of the screen background, and the highlighting of input fields. This phase is optional; you can instead begin your form definition with the required Layout phase by typing LAYOUT (or simply L) and pressing the RETURN key.

3.2.2 The Layout Phase

In the Layout phase, you create a screen image of the form by typing in the background text and field identifiers. **Background text** is displayed on the screen whenever the form is displayed; it identifies the information that the user is to type on the form. **Field identifiers** show the locations of fields where data can be displayed or entered on a form.

When you enter the Layout phase, FDU clears your screen and displays a cursor status line on the bottom line of the screen. The status line initially looks like this:

```
Cursor TXT NOR Line  1 Column   1 Modes TXT OVS
```

When you begin creating the form layout, the cursor is positioned at the upper left corner of the screen, line 1, column 1. As you type, the line and column indicators in the cursor status line change as the cursor moves. When creating a simple form, you need not be concerned about the cursor setting. Mode settings are described later in this section.

You can now type background text on the form and specify the contents of each form field. You use the following field identifiers to describe the most common types of characters a field can contain:

- A -- alphabetic characters only (the letters A-Z and a-z)
- 9 -- numeric characters only (the numbers 0-9)
- X -- any displayable character (alphabetic, numeric, and special, such as * and %)

Use as many field identifiers as necessary to describe the length of the field. A field can also contain certain punctuation marks, such as a decimal point or a slash. When your program displays the form, the field identifiers do not appear; rather, the input typed by the user (and any punctuation included in the field definition) appears in the field.

The form editor must be able to distinguish between the characters you type as background text and the characters you type to denote field identifiers. The cursor status line shows whether you are typing background text (Text mode, indicated by TXT) or field identifiers (Field mode, indicated by FLD). To switch from Text to Field mode, press the GOLD key followed by 8 on the keypad (the GOLD-keypad 8 combination); then type the field identifiers. To switch back to Text mode, press the keypad 8 key. The mode settings on the cursor status line change as you switch between Text and Field modes.

Like many forms, the personnel form contains a date field. TDMS provides a simple way to define such a field and offers you a choice of several date formats. To insert a date field on a form, make sure you are in Field mode and then press the

GOLD-D key combination. The cursor status line is replaced by the following four date formats:

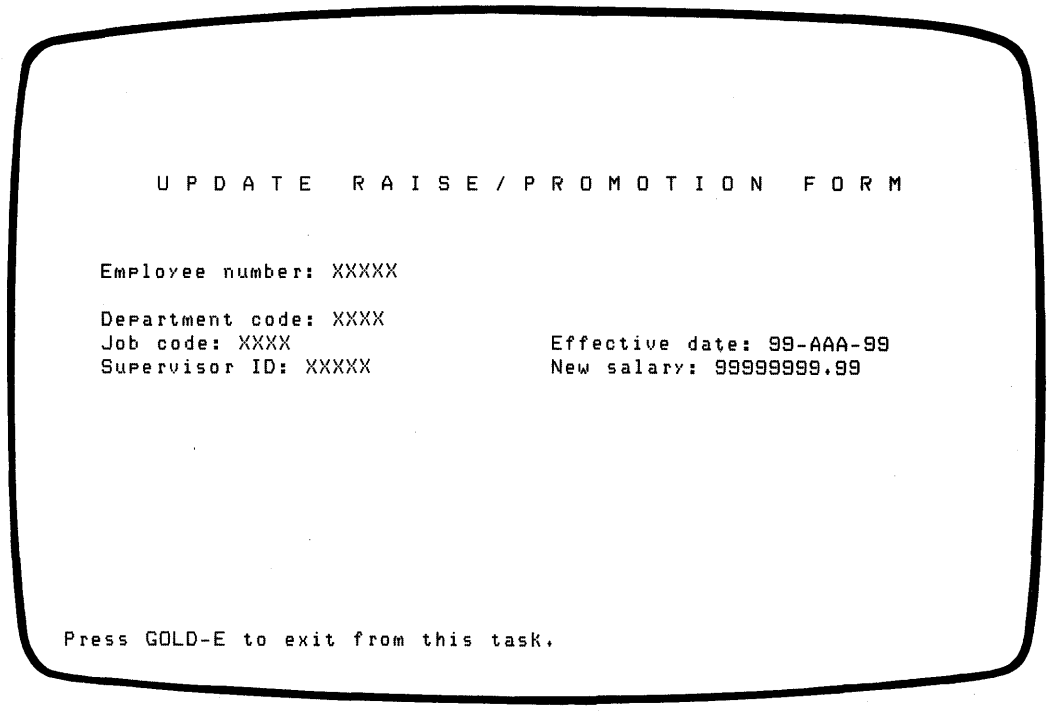
1. Month Day, Year (AAA 99, 9999)
2. Day-Month-Year (99-AAA-99)
3. Month/Day/Year (99/99/99)
4. Day-Month-Year (99-99-99)

You are asked to choose a format by typing the corresponding number. The form editor automatically inserts the appropriate field identifiers on the form. When the form is displayed as part of an application, TDMS displays the current date in the date field by default.

To position the background text and the input fields where you want them on the form, you can use the RETURN key, the SPACE bar, and the arrow keys to align text and fields and put blank lines between entries. If you need to change text or fields while in the Layout phase, you can either overstrike the characters on the screen or insert new characters in front of existing ones. The cursor status line shows whether you are overstriking characters (Overstrike mode, indicated by OVS) or inserting characters (Insert mode, indicated by INS). To switch from Overstrike to Insert mode, press the GOLD-PF3 key combination; to switch back to Overstrike mode, press the PF3 key. The mode settings on the cursor status line change as you switch between Overstrike and Insert modes.

Figure 3-2 shows the form layout for the personnel form. It contains the employee number field, which the user supplies in the inquiry operation, and the job code, supervisor's employee number, and salary fields, which the user can modify in the update operation. It also contains the other fields in the job history and salary history relations, since you need to store new records in these relations when an employee receives raise or a promotion. Note that on the form shown in Figure 3-2, the second date format is used for the starting date field.

The last line on the form is background text that reminds the user to press the GOLD-E key combination to exit from the program without completing the update; such a key combination is called a **program request key**. You define a program request key in a request to allow the user some control over a running application (see Section 3.3.1). When you are finished creating a form layout, press the GOLD-keypad 7 key combination. This ends the Layout phase and returns you to the Phase Selection Menu.



ZK-00047-00

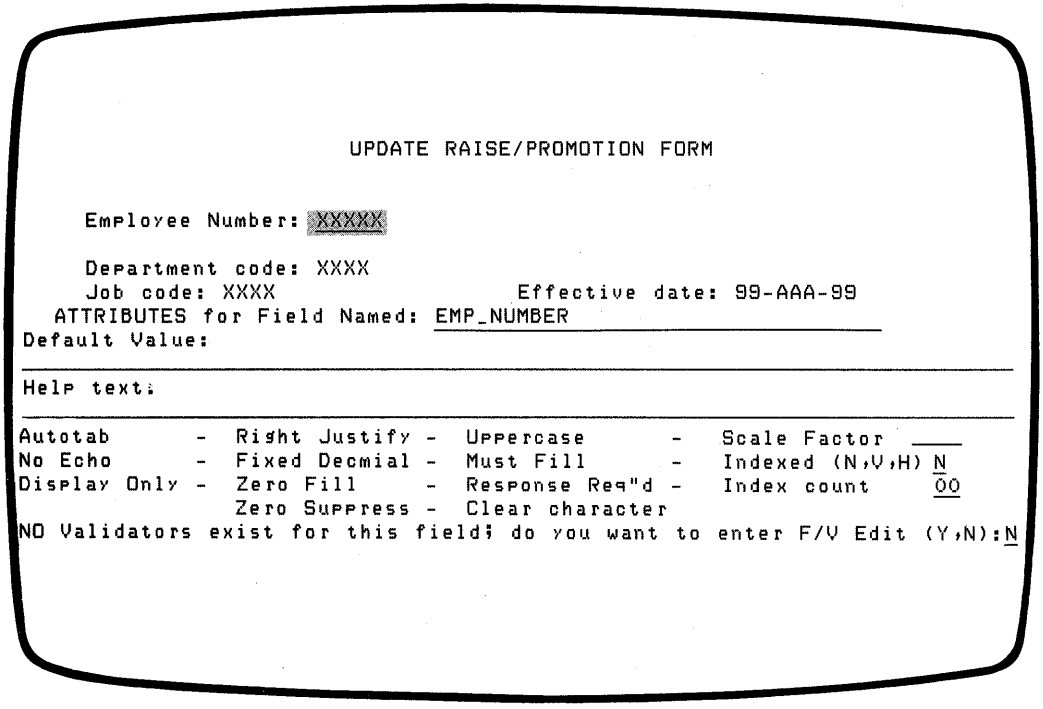
Figure 3-2: Personnel Form Layout

3.2.3 The Assign Phase

After the Layout phase, you select the Assign phase to provide names for the form fields and such optional information as default values and help messages. To enter the Assign phase, type ASSIGN (or A) at the Phase Selection Menu and press RETURN. When the Assign Phase Menu is displayed, type 2 to indicate that you want to assign attributes to all fields, and then press RETURN. If you forgot to switch to field mode when you typed the field identifiers, FDU reports that the form contains no fields; in other words, the form consists entirely of background text. To correct this error, select the Layout phase from the Phase Selection Menu; then delete the field identifiers you typed, switch to field mode with the GOLD-keypad 8 key combination, and retype the field identifiers.

When you enter the Assign phase, the Attribute Assignment Form is displayed, and the field identifier for the first field in the form is shown bolded, underlined, and blinking. In the Attributes section of the form, the field name for the first field is F$0001. TDMS assigns this name by default to the first field, and subsequent fields are numbered consecutively (F$0002, F$0003, and so on). Each field must have a unique field name. You can accept the default field name if you like, but you should instead choose a more descriptive field name. To delete the default name, press the LINE FEED key and type the name of your choice.

Figure 3-3 shows how the Attribute Assignment Form might look after the first field has been renamed to EMP NUMBER.



ZK-00048-00

Figure 3-3: Attribute Assignment Form for Personnel Form

You can now move to the other entries on the Attribute Assignment Form by pressing the TAB key. The next two entries allow you to provide a default value for a field and help text to inform the user what kind of data is required. The remaining entries let you select restrictions on the input and display of data on

the form. If you want to accept all the defaults, press RETURN after you change the field name. If you want to change a default, press the TAB key enough times to position the cursor on the entry you want to change; then mark the entry with an X.

To prevent the user from modifying a displayed field, choose the Display Only characteristic when you assign the field's name. At run time, the cursor skips over display-only fields, preventing the user from trying to change their contents. For the personnel form, the job code and effective date (which is the current date) must be displayed so that they can be stored in new job history and salary history records. You should define these fields as display-only so that the user can see them and they can be stored in the database but they cannot be changed.

To end the assignment of attributes for a field at any time, press RETURN.

You can give form fields any meaningful names you like; for the purposes of illustration in the rest of this chapter, the fields on the update form have the following names:

- EMP_NUMBER
- DEPT_CODE
- JOB_CODE
- JOB_START
- SUPERVISOR_ID
- SALARY

Field names are used in TDMS request definitions that display the form. When you have assigned field names and any other attributes you need for all the fields on your form, you end the Assign phase of the form editor by pressing the GOLD-keypad 7 key combination. The Phase Selection Menu reappears, and you can select another phase.

3.2.4 The Order Phase

The next phase of form development is the Order phase, where you can change the order in which users enter data on the form. By default, form fields are filled from left to right and from top to bottom. Because this order is practical for most applications, you can usually skip the Order phase if you typed the field identifiers in this order in the Layout phase.

3.2.5 The Exit Phase

To leave the form editor, choose the Exit phase by typing EXIT (or E) at the Phase Selection Menu and pressing RETURN. FDU then asks whether you want to save the form in the CDD. To save the form, press RETURN. FDU stores the

form definition in the CDD under the name you specified with the CREATE FORM command. To leave FDU, type EXIT or press CTRL/Z at the FDU> prompt.

To see the definition of a form stored in the CDD, you use the LIST FORM command in FDU. This command shows the background text, field identifiers, and attribute information about a form definition. You can either display the command output on your terminal or write it to a file. For example:

```
FDU> LIST FORM PERS_UPDATE_RAISEPRO_FORM /OUTPUT=PERS_RP_FORM.TXT
```

This command writes the definition of PERS_UPDATE_RAISEPRO_FORM in a file named PERS_RP_FORM.TXT in your default VMS directory.

3.3 Defining Requests

After you define the forms for your application, you can define the requests that move the user's input from form fields to workspace fields and move the program's output from the workspace back to the form. You define a request with the Request Definition Utility, or RDU. The easiest way to use RDU is to create a file of RDU instructions with a text editor, such as EDT, and then submit it as a command file to RDU, which checks for possible errors. The *VAX TDMS Request Manual* describes RDU in detail.

For the update program, you need two requests: a retrieval request to display the form and collect an employee number and an update request to display the form with information obtained from the database and collect the user's changes. These requests are explained in the following sections.

3.3.1 Defining the Retrieval Request

A simple request contains two parts, a header and a base. The header identifies the TDMS form definitions and the workspace definitions needed in the request. The base contains mapping and usage instructions for TDMS to use whenever an application program calls the request.

3.3.1.1 The Request Header -- In the header, you use FORM IS and RECORD IS instructions to identify the CDD locations of your form and workspace definitions. You can explain the purpose of the request by including comments within the delimiters /* and */ in a DESCRIPTION instruction. The retrieval request displays the personnel form, PERS_UPDATE_RAISEPRO_FORM. It also needs two workspaces, one to hold the employee number that the user types and one to hold miscellaneous information that the program needs but that is not stored in the database itself.

A workspace must have a record definition stored in the CDD; you can usually use a record definition that was stored in the CDD when you defined your DBMS or Rdb/VMS database. To decide which definition to use for the workspace that holds the employee number, consider which relations in the personnel database contain that field and which are relevant to the records you need to update. The current job and salary information is stored in two database relations, JOB_HISTORY and SALARY_HISTORY, both of which contain an employee number field. You can use either JOB_HISTORY or SALARY_HISTORY to define the workspace. To define the workspace of miscellaneous information, called PERS_WORKSPACE in this application, you define a CDD record with the CDD Data Definition Language, described in Section 3.4.

The following example shows the header of the retrieval request. Because of the CDD hierarchy created when you define a database, you should specify the complete CDD path name rather than a given name if you use a DBMS or an Rdb/VMS record definition for a workspace.

```
FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
RECORD IS CDD$TOP.RDBPERS.PERS_WORKSPACE;

DESCRIPTION /* Accept the employee ID number for retrieving
               job and salary information for raise/promotion
               update */;
```

3.3.1.2 The Request Base -- The RDU instructions in the base of a request are performed every time an application program calls the request. The instructions can appear in any order because TDMS has a predefined order for executing them.

The main purpose of most requests is to display a form and either collect input or display output. In the base, you specify which form is to be displayed and which workspace fields are to be assigned the values typed on the form. You also use the base to define program request keys and do error handling.

The USE FORM instruction displays a form as it looked when the previous call to the request ended; that is, the background text and any values that had been supplied for form fields are shown on the form. If the form was not displayed in the previous request call, TDMS displays the form with the background text and any defaults established in the form definition. The following instruction displays the personnel form:

```
USE FORM PERS_UPDATE_RAISEPRO_FORM;
```

Once the form is displayed, the user can type input at the terminal. To direct TDMS to collect input from the form and store it in a workspace, you use an INPUT TO instruction, naming each form field and the corresponding workspace field. When the user has filled in all the required fields, TDMS copies the value

from the form into the workspace. This INPUT TO instruction collects a value from the EMP_NUMBER field on the form and stores it in the EMPLOYEE_ID field in a workspace:

```
INPUT EMP_NUMBER TO EMPLOYEE_ID;
```

In the personnel form shown in Figure 3-2, the last line reminds the user to press the GOLD-E program request key to exit from the program. The user might choose to exit if, for example, an error too severe for the user to correct occurred during a data manipulation operation.

You define a program request key in the base of the request definition, using the PROGRAM KEY IS instruction. You must specify a field in a workspace stored in the CDD and state what value will be returned to the field when the user presses the program request key. The program can then test the value of the field and take the appropriate action. You define a program request key to exit from the program as follows:

```
PROGRAM KEY IS GOLD "E"
   NO CHECK;
   RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
END PROGRAM KEY;
```

PROGRAM_REQUEST_KEY is a field in PERS_WORKSPACE that the program tests to determine what action the user wants to take. The NO CHECK modifier causes TDMS to terminate the request without executing any outstanding instructions when the user presses the program request key.

Some errors are predictable in a database application; for example, the record that a user wants to modify might be locked by another user, or the user might type an employee number that does not exist in the database. When an error does occur, you should notify the user by displaying an error message on the screen. To do so, the program must first store an error value in a field of a workspace that the request can access. The program then recalls the request, which tests the value in the field and redisplays the form with an error message on the bottom line. The user can then retry the operation or exit from the application by pressing the program request key.

In the update program, error values are stored in a control field called ERROR_FIELD, which is contained in PERS_WORKSPACE. The error values are character strings that briefly indicate the type of error; for example, "LOCKED" indicates a locked record, and "NOTFND" indicates a nonexistent employee number. In the request, you use a CONTROL FIELD IS instruction to test whether ERROR_FIELD contains either of these values. Within this instruction, you use MESSAGE LINE IS instructions to associate each value with an

appropriate error message. The following example shows the CONTROL FIELD IS instruction for the retrieval request:

```
CONTROL FIELD IS ERROR_FIELD
   "LOCKED" : MESSAGE LINE IS
              "Record is locked. Try again or exit with GOLD-E.";
   "NOTFND" : MESSAGE LINE IS
              "Employee not found. Try another number or exit with GOLD-E.";
END CONTROL FIELD;
```

You complete the request with the END DEFINITION instruction.

TDMS executes CONTROL FIELD IS instructions before any other instructions in a request. It then executes any output operations and finally any input operations, including the evaluation of program request keys. Example 3-1 shows the complete request definition for the retrieval request.

```
FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
RECORD IS PERS_WORKSPACE;

DESCRIPTION /* Accept the employee ID number for retrieving
               job and salary information for raise/promotion
               update */;

USE FORM PERS_UPDATE_RAISEPRO_FORM;

INPUT EMP_NUMBER TO EMPLOYEE_ID;

PROGRAM KEY IS GOLD "E"
   NO CHECK;
   RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
END PROGRAM KEY;

CONTROL FIELD IS ERROR_FIELD
   "LOCKED" : MESSAGE LINE IS
              "Record is locked. Try again or cancel with GOLD-E.";
   "NOTFND" : MESSAGE LINE IS
              "Employee not found. Try another number or cancel with GOLD-E.";
END CONTROL FIELD;

END DEFINITION;
```

Example 3-1: Retrieval Request Definition

3.3.2 Defining the Update Request

After the application program calls the retrieval request, it retrieves data from the database and stores it in a workspace. The update request displays the retrieved data and lets the user make any necessary changes. The program can then update the database accordingly. Like the retrieval request, the update request displays the personnel form and uses PERS_WORKSPACE to store the

values of the program request key and the control field. It uses JOB_HISTORY to define a workspace for an employee's job history information and SALARY_HISTORY to define a workspace for salary history information. Thus, the header for the update request is:

```
FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;
RECORD IS PERS_WORKSPACE;

DESCRIPTION /* Display job and salary information and accept
               changes to indicate a raise and/or a promotion */;
```

You use a USE FORM instruction to display the form in the update request:

```
USE FORM PERS_UPDATE_RAISEPRO_FORM;
```

To transfer fields in a workspace to a form, you use an OUTPUT TO instruction. You can then transfer the modified form fields back to the workspaces with an INPUT TO instruction.

In some cases, you want to display the value of a field for the user's benefit but you do not want to let the user enter data in the field. You can accomplish this in two ways: you can display the value in an OUTPUT TO instruction but not accept input for it in an INPUT TO instruction, or you can use the RETURN TO instruction. The RETURN TO instruction transfers the value in a form field to a field in a workspace without positioning the cursor in the field and thus giving the user an opportunity to change the value. However, the value must have been stored in the field by some default method, not by the OUTPUT TO instruction. For example, TDMS automatically displays the current date in a field with a date format that is not listed in an OUTPUT TO instruction. With a RETURN TO instruction, you can transfer the current date to a field in a workspace.

The update request uses these three instructions:

```
OUTPUT DEPARTMENT_CODE          TO DEPT_CODE,
       JOB_CODE                 TO JOB_CODE,
       SUPERVISOR_ID            TO SUPERVISOR_ID,
       SALARY_AMOUNT            TO SALARY;

INPUT JOB_CODE      TO JOB_CODE,
      SUPERVISOR_ID TO SUPERVISOR_ID,
      SALARY        TO SALARY_AMOUNT;

RETURN JOB_START TO JOB_START;
```

Because DEPT_CODE is a Display Only field in the form definition, you cannot direct TDMS to accept an input value for it; thus, the user cannot attempt to change its value. Likewise, the value of JOB_START is the current date, and the user cannot attempt to change it.

Like the retrieval request, the update request should define a control field for testing errors and a program request key to let the user exit easily from the application. Because you can expect locked record and nonexistent record errors to occur when the program tries to update the database, you can use the same control field definition in the update request as in the retrieval request. You can also use the same program request key definition so that the terminal user can exit from the application in the same way from both requests.

You end the update request definition with the END DEFINITION instruction.

Example 3-2 shows the complete update request definition.

```
FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
RECORD IS CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;
RECORD IS PERS_WORKSPACE;

DESCRIPTION /* Display job and salary information and accept
               changes to indicate a raise and/or a promotion */;

USE FORM PERS_UPDATE_RAISEPRO_FORM;

OUTPUT JOB_CODE        TO JOB_CODE,
       DEPARTMENT_CODE TO DEPT_CODE,
       SUPERVISOR_ID   TO SUPERVISOR_ID,
       SALARY_AMOUNT   TO SALARY;

INPUT JOB_CODE      TO JOB_CODE,
      SUPERVISOR_ID TO SUPERVISOR_ID,
      SALARY        TO SALARY_AMOUNT;

RETURN JOB_START TO JOB_START;

PROGRAM KEY IS GOLD "E"
   NO CHECK;
   RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
END PROGRAM KEY;

CONTROL FIELD IS ERROR_FIELD
   "LOCKED" : MESSAGE LINE IS
              "Record is locked. Try again or cancel with GOLD-E.";
   "NOTFND" : MESSAGE LINE IS
              "Employee not found. Try another number or cancel with GOLD-E.";
END CONTROL FIELD;

END DEFINITION;
```

Example 3-2: Update Request Definition

3.4 Defining Workspaces

Most application programs use program request keys or control fields to control a running application. These fields are stored not in a database but in a workspace whose definition resides in the CDD. With the CDD Data Definition Language Utility (CDDL), you define a record to use as a workspace and enter the definition

directly in the CDD. The easiest way to use CDDL is to create a source file of CDDL statements with a text editor, such as EDT, and then submit the file to the CDDL compiler, which checks for possible errors. The *VAX Common Data Dictionary Data Definition Language Reference Manual* contains complete reference information on CDDL.

3.4.1 Defining the Record

A record definition begins with a DEFINE statement and ends with an END statement. The DEFINE statement gives the name of the record, which can be either the full CDD path name, starting with CDD$TOP, or the definition's given name. After the DEFINE statement, you can use an optional DESCRIPTION statement to include comments in the record definition.

The body of a record definition is a field description statement that lists all the fields in the record and describes their characteristics, including the data type and size. Among the kinds of fields a CDD record can have are elementary fields and STRUCTURE fields. An elementary field is not divided into subordinate fields, while a STRUCTURE field is further subdivided. You usually define a CDD record as a STRUCTURE field composed of elementary fields and other STRUCTURE fields. You can describe individual fields by enclosing comments within the delimiters /* and */.

Example 3-3 shows the definition of the workspace used in the update program.

```
DEFINE RECORD PERS_WORKSPACE
   DESCRIPTION IS /* Miscellaneous fields */.

PERS_WORKSPACE STRUCTURE.
   PROGRAM_REQUEST_KEY              DATATYPE TEXT SIZE 6
                                    INITIAL_VALUE IS "      ".
   ERROR_FIELD                      DATATYPE TEXT SIZE 6
                                    INITIAL_VALUE IS "      ".
   NOT_FOUND                        DATATYPE TEXT SIZE 1
                                    INITIAL_VALUE IS " ".
   SAL_AMT                          DATATYPE SIGNED LONGWORD.
   JOB                              DATATYPE TEXT SIZE 4
                                    INITIAL_VALUE IS "    ".
   TEST_FIELD                       DATATYPE TEXT SIZE 1
                                    INITIAL_VALUE IS " ".
END PERS_WORKSPACE STRUCTURE.

END PERS_WORKSPACE.
```

Example 3-3: PERS_WORKSPACE Definition in the CDD

Most of the fields in this workspace are defined as TEXT and initialized with a value of spaces. The SIZE information specifies the maximum number of characters that the field's value can have. The NOT_FOUND field is used to indicate that a record with the specified employee number does not exist in the database. The SAL_AMT and JOB_CODE fields are used to hold an employee's present salary and job code so that the update program can determine whether the employee

received a raise, a promotion, or both. JOB_CODE is declared as a signed longword for more efficient storage. TEST_FIELD is defined for the ACMS personnel application, which also uses this workspace; the application is described in Section 4.1 and contained in Section A.1.

3.4.2 Inserting the Record Definition in the CDD

After you define the workspace, you must submit the definition to the CDDL utility to be compiled. As it compiles the source file, CDDL reports any errors that it finds. If it finds none, it inserts the definition in your default CDD directory (or in the directory you specified in the DEFINE RECORD statement). To use the CDDL compiler, you should first define CDDL as a global symbol in your login command file or at DCL command level:

```
$ CDDL :== $CDDL
```

If you stored the definition for PERS_WORKSPACE in a source file named PERS_WORKSPACE.DDL, you would compile it with the following command:

```
$ CDDL/AUDIT PERS_WORKSPACE
```

Because .DDL is the default file type for CDDL source files, you need not specify it on the command line. The /AUDIT qualifier creates a history list for PERS_WORKSPACE and records the date and time of its insertion in CDD$TOP.RDBPERS.

If CDDL finds errors when compiling your source file, it displays error messages on your screen. In addition, CDDL automatically creates a listing file that contains the source text and the error messages for any errors it found. The listing file is created in your default VMS directory with the same file name as the source file and the file type .LIS. If you need to correct errors or change a definition already stored in the CDD, you can edit the source file and then recompile it. using the CDDL command with the /REPLACE qualifier to insert the changed definition in the CDD. For example:

```
$ CDDL/REPLACE/AUDIT PERS_WORKSPACE
```

This command recompiles the PERS_WORKSPACE.DDL source file (which you have presumably edited since you first created the PERS_WORKSPACE definition) and replaces the existing definition with your changed version.

3.5 Storing Request Definitions in the CDD

After you have defined all the forms, requests, and workspaces you need, you store the request definitions in the CDD with RDU's CREATE REQUEST or REPLACE REQUEST command. Both commands check for syntax errors and, if no errors are found, store the definitions in the CDD. The REPLACE REQUEST command works exactly like CREATE REQUEST if the definition does not exist.

Just in case a request definition does not compile correctly the first time you submit it to RDU, you should always use the REPLACE REQUEST command; then you do not have to remember to change the command in your command file when you correct the other errors.

To submit a definition to RDU, include either command as the first line in the command file and specify the request definition's complete path name or given name in the CDD. For example, the following command names the retrieval request:

```
REPLACE REQUEST PERS_GET_RAISEPRO_REQUEST
```

You can name the update request with a similar command:

```
REPLACE REQUEST PERS_PUT_RAISEPRO_REQUEST
```

When RDU processes either of these commands, it checks the rest of the command file and, if it finds no errors, creates the request definition in the CDD. If your default CDD directory is set to the directory where you want to store your definitions (in this case, CDD$TOP.RDBPERS), you can use just the given name in the REPLACE command.

To enter RDU, first define RDU as a global symbol in your login command file or at DCL command level:

```
$ RDU :== $RDU
```

Then, to invoke RDU, simply type RDU. At the RDU> prompt, you can submit your command file to RDU and insert your request definition in the CDD. If you stored the retrieval request in a file named PERS_GET_REQUEST.COM, you would submit it to RDU as follows:

```
$ RDU
RDU>@PERS_GET_REQUEST
```

Because .COM is the default file type for RDU command files, you need not specify it on the command line. Similarly, you would submit the update request in a command file named PERS_PUT_REQUEST.COM as follows:

```
RDU>@PERS_PUT_REQUEST
```

RDU generates warning messages when you insert the retrieval and update requests in the CDD because TDMS does not support the INITIAL_VALUE clause used in the definition of PERS_WORKSPACE. TDMS simply ignores this clause, so you can ignore the message. If RDU detects other errors in your request definition, you must edit the command file and resubmit it. You must repeat these two steps until RDU reports that it processed the file without errors.

A common error is that the form field names you use in the request do not match the names you assigned to the fields in the form definition, or that the record field names you use in the request do not match the names you used in the record definition. Be sure that all the field names in the request definition are correct before you submit the command file to RDU.

To exit from RDU, use the EXIT command or press CTRL/Z.

3.6 Defining and Building a Request Library

The TDMS requests you use in an application program must be stored in a request library whose definition resides in the CDD. You define a request library with RDU instructions, which you can submit to RDU in a command file; if RDU finds no errors, it inserts the request library definition in the CDD.

After you define the request library, you must convert it into a request library file. This file is a VAX RMS file that contains binary versions of the requests and other information about the forms and records they use. At run time, TDMS must execute the requests' instructions in their binary form rather than as RDU source instructions.

You define the request library with REQUEST IS instructions that list program. To submit the definition to RDU and store it in the CDD, you add the CREATE LIBRARY or REPLACE LIBRARY command at the top of the definition, specifying the request library's complete path name or given name in the CDD. If your default CDD directory is set to the correct directory (in this case, CDD$TOP.RDBPERS), you can use just the given name.

Example 3-4 shows the request library definition for the update program.

```
REPLACE LIBRARY PERS_REQLIB
   REQUEST IS PERS_GET_RAISEPRO_REQUEST;
   REQUEST IS PERS_PUT_RAISEPRO_REQUEST;
END DEFINITION;
```

Example 3-4: Request Library Definition for Update Program

If you stored the request library definition in a file named PERS_REQLIB.COM, you would submit it to RDU as follows:

```
$ RDU
RDU>@PERS_REQLIB
```

Once the request library is stored in the CDD, you can build the request library file with RDU's BUILD LIBRARY command. You must specify the request library's path name or given name in the CDD and the VMS file specification you want the library file to have. For example:

```
RDU>BUILD LIBRARY PERS_REQLIB PERS$EXE:PERS_REQLIB.RLB
```

In this example, RDU uses the request library definition PERS_REQLIB to locate the requests in your default CDD directory and place them in the new request library file in the VMS directory referred to by the logical name PERS$EXE. The request library file is sometimes called the .RLB file because .RLB is the default file type.

Because TDMS does not support the INITIAL_VALUE clause used in the definition of PERS_WORKSPACE, RDU generates warning messages when you build the request library file for the update program. These messages appear at every reference to PERS_WORKSPACE; you can ignore them.

Before you build a request library file, be sure that:

- All the requests in the request library definition are defined in the CDD
- All the forms and records referred to in the requests are defined in the CDD
- Your default CDD directory is set to the directory where the request library, request, form, and record definitions are stored if you did not specify complete CDD path names in the definitions

RDU issues error messages and does not build a request library file if any of these conditions are not met. If the BUILD command fails, you must correct the errors and resubmit the request library definition to RDU, repeating these two steps until the definition is processed without errors.

For more information about defining and building request libraries, see the *VAX TDMS Request Manual.*

3.7 TDMS Application Programming

Application programs use TDMS programming calls to handle the interactions with the user's terminal. For most programs, the following calls are sufficient to handle input and output requirements:

- TSS$OPEN_RLB to open the request library file
- TSS$OPEN to open a channel to the terminal
- TSS$REQUEST to call a request and execute the instructions it contains
- TSS$CLOSE_RLB to close the request library file
- TSS$CLOSE to close the channel
- TSS$SIGNAL to signal the return status of an unsuccessful programming call

The update program described in this chapter uses all of these calls and issues two calls to TSS$REQUEST. The first calls the retrieval request, PERS_GET_RAISEPRO_REQUEST, to display the personnel form. After the user fills in the employee number field, the program uses DML statements to start a transaction, retrieve information from the JOB_HISTORY and SALARY_HISTORY relations in the database, and end the transaction. It saves the current job code and salary in two fields of PERS_WORKSPACE and then issues the second call to TSS$REQUEST, which calls the update request.

PERS_PUT_RAISEPRO_REQUEST redisplays the form and lets the user make changes to the job and salary data. The program then starts another transaction, determines whether the employee received a raise, a promotion, or both, and uses more DML statements to add ending dates (if necessary) to the current JOB_HISTORY and SALARY_HISTORY records. Finally, the program stores new records in these relations for the employee's new job and salary and commits these changes to the database. Program control returns to the first TSS$REQUEST call, and the user can enter another employee number or use the program request key to exit from the program. The *VAX TDMS Programming Manual* explains the TDMS programming calls in detail.

Example 3-5 shows the complete program that updates the personnel database for an employee's raise and promotion. To compile this program, you must use the Rdb/VMS precompiler for COBOL. The precompiler first checks the syntax of the DML statements in your procedure and converts these statements into equivalent calls to the database. It then invokes the COBOL compiler to check the syntax of your COBOL statements and generate object code for your procedure. The compiler generates warning messages for the use of the DATE data type in the database definition; COBOL must convert the DATE data type into an equivalent numeric string. You can ignore these messages.

After you compile the program, you must link the object module with the VAX Linker. The warning-level errors reported for the DATE data type are also detected when you link the object module, but they do not affect the execution of your program.

You should use the /DEBUG qualifier when you compile and link the program so that you can debug it with the VAX Symbolic Debugger. See the *VAX Rdb/VMS Guide to Programming* for information on compiling COBOL programs with embedded DML statements. See the *VAX COBOL User's Guide* for information on compiling COBOL programs and interpreting COBOL error messages.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    RAISEPRO.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
WORKING-STORAGE SECTION.

    &RDB& INVOKE DATABASE FILENAME 'PERS$EXE:PERSONNEL'

01 CHANNEL                      PIC S9(5) COMP.
01 STATUS-RESULT                PIC S9(5) COMP.
01 REQUEST-LIBRARY-FILE         PIC X(20)
                                  VALUE IS "PERS$EXE:PERSLIB.RLB".
01 LIBRARY-ID                   PIC S9(5) COMP.
01 REQUEST1                     PIC X(25)
                                  VALUE IS "PERS_GET_RAISEPRO_REQUEST".
01 REQUEST2                     PIC X(25)
                                  VALUE IS "PERS_PUT_RAISEPRO_REQUEST".
01 CLEAR-SCREEN                 PIC S9(5) COMP
                                  VALUE IS 1.
01 REC-LOCKED                   PIC X(6)
                                  VALUE IS "LOCKED".
01 REC-NOT-FOUND                PIC X(6)
                                  VALUE IS "NOTFND".
01 RDB$_DEADLOCK                PIC S9(9) COMP
                                  VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT           PIC S9(9) COMP
                                  VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL                   PIC S9(9) COMP
                                  VALUE IS EXTERNAL LIB$SIGNAL.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_SCREEN. ==.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_SCREEN. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION GIVING STATUS-RESULT.

MAIN SECTION.
010-OPEN.

    SET STATUS-RESULT TO SUCCESS.

    CALL "TSS$OPEN_RLB" USING
         BY DESCRIPTOR REQUEST-LIBRARY-FILE,
```

Example 3-5: Update Program Using TDMS Calls

```
        BY REFERENCE LIBRARY-ID
        GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

    CALL "TSS$OPEN" USING
        CHANNEL
        GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

020-RETRIEVE-INFO.

    CALL "TSS$REQUEST" USING
        BY REFERENCE CHANNEL,
        BY REFERENCE LIBRARY-ID,
        BY DESCRIPTOR REQUEST1,
        BY REFERENCE JOB_HISTORY_SCREEN,
                     PERS_WORKSPACE
        GIVING STATUS-RESULT.

    MOVE "SUCCES" TO ERROR_FIELD.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

    IF PROGRAM_REQUEST_KEY = "EXIT"
    THEN
        GO TO 040-CLEANUP.

    &RDB& START_TRANSACTION READ_ONLY
    &RDB&    ON ERROR
               PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
               &RDB& ROLLBACK
               GO TO 020-RETRIEVE-INFO
    &RDB&    END_ERROR

    MOVE "T" TO NOT_FOUND.

    &RDB& FOR JH IN JOB_HISTORY WITH
    &RDB&    JH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_SCREEN
    &RDB&    AND JH.JOB_END MISSING
    &RDB&      ON ERROR
                 PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
                 &RDB&  ROLLBACK
                 GO TO 020-RETRIEVE-INFO
    &RDB&      END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&    GET
    &RDB&      ON ERROR
                 PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
```

(continued on next page)

Example 3-5: Update Program Using TDMS Calls (Cont.)

```
            &RDB&  ROLLBACK
            GO TO 020-RETRIEVE-INFO
&RDB&     END_ERROR
&RDB&       JOB_CODE IN JOB_HISTORY_SCREEN = JH.JOB_CODE;
&RDB&       DEPARTMENT_CODE IN JOB_HISTORY_SCREEN = JH.DEPARTMENT_CODE;
&RDB&       SUPERVISOR_ID IN JOB_HISTORY_SCREEN = JH.SUPERVISOR_ID
&RDB&   END_GET
&RDB& END_FOR

IF NOT_FOUND = "T"
THEN
&RDB&   ROLLBACK
        MOVE REC-NOT-FOUND TO ERROR_FIELD
        GO TO 020-RETRIEVE-INFO.

MOVE JOB_CODE OF JOB_HISTORY_SCREEN TO JOB OF PERS_WORKSPACE.

MOVE "T" TO NOT_FOUND.

&RDB& FOR SH IN SALARY_HISTORY WITH
&RDB&   SH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_SCREEN
&RDB&   AND SH.SALARY_END MISSING
&RDB&     ON ERROR
            PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
            &RDB&  ROLLBACK
            GO TO 020-RETRIEVE-INFO
&RDB&     END_ERROR

MOVE "F" TO NOT_FOUND

&RDB&   GET
&RDB&     ON ERROR
            PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
            &RDB&  ROLLBACK
            GO TO 020-RETRIEVE-INFO
&RDB&     END_ERROR
&RDB&       SALARY_AMOUNT IN SALARY_HISTORY_SCREEN = SH.SALARY_AMOUNT
&RDB&   END_GET
&RDB& END_FOR

IF NOT_FOUND = "T"
THEN
&RDB&   ROLLBACK
        MOVE REC-NOT-FOUND TO ERROR_FIELD
        GO TO 020-RETRIEVE-INFO.

MOVE SALARY_AMOUNT OF SALARY_HISTORY_SCREEN TO
    SAL_AMT OF PERS_WORKSPACE.

&RDB& COMMIT.

030-UPDATE-INFO.
    CALL "TSS$REQUEST" USING
         BY REFERENCE CHANNEL,
         BY REFERENCE LIBRARY-ID,
```

Example 3-5: Update Program Using TDMS Calls (Cont.)

```
        BY DESCRIPTOR REQUEST2,
        BY REFERENCE JOB_HISTORY_SCREEN,
                     SALARY_HISTORY_SCREEN,
                     PERS_WORKSPACE
        GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

    IF PROGRAM_REQUEST_KEY = "EXIT"
    THEN
        GO TO 040-CLEANUP.

    &RDB& START_TRANSACTION READ_WRITE RESERVING
    &RDB&       JOB_HISTORY, SALARY_HISTORY, EMPLOYEES FOR SHARED WRITE
    &RDB&     ON ERROR
                PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
                &RDB& ROLLBACK
                GO TO 020-RETRIEVE-INFO
    &RDB&     END_ERROR

    MOVE "T" TO NOT_FOUND.

    IF JOB OF PERS_WORKSPACE = JOB_CODE OF JOB_HISTORY_SCREEN
    THEN
        MOVE "F" TO NOT_FOUND
    ELSE
    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&     EMPLOYEE_ID IN JOB_HISTORY_SCREEN
    &RDB&     AND JH.JOB_END MISSING
    &RDB&       ON ERROR
                  PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
                  &RDB&  ROLLBACK
                  GO TO 030-UPDATE-INFO
    &RDB&       END_ERROR

          MOVE "F" TO NOT_FOUND

    &RDB&     MODIFY JH USING
    &RDB&       ON ERROR
                  PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
                  &RDB&  ROLLBACK
                  GO TO 030-UPDATE-INFO
    &RDB&       END_ERROR
    &RDB&         JH.JOB_END = JOB_START IN JOB_HISTORY_SCREEN
    &RDB&     END_MODIFY
    &RDB& END_FOR

    IF NOT_FOUND = "T"
    THEN
    &RDB&     ROLLBACK
              MOVE REC-NOT-FOUND TO ERROR_FIELD
              GO TO 030-UPDATE-INFO
    END-IF
```

(continued on next page)

Example 3-5: Update Program Using TDMS Calls (Cont.)

```
&RDB& STORE JH IN JOB_HISTORY USING
&RDB&   ON ERROR
          PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
          &RDB&   ROLLBACK
          GO TO 030-UPDATE-INFO
&RDB&   END_ERROR
&RDB&     JH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_SCREEN;
&RDB&     JH.JOB_CODE = JOB_CODE IN JOB_HISTORY_SCREEN;
&RDB&     JH.DEPARTMENT_CODE = DEPARTMENT_CODE IN JOB_HISTORY_SCREEN;
&RDB&     JH.JOB_START = JOB_START IN JOB_HISTORY_SCREEN;
&RDB&     JH.SUPERVISOR_ID = SUPERVISOR_ID IN JOB_HISTORY_SCREEN
&RDB& END_STORE
END-IF.

MOVE "T" TO NOT_FOUND.

IF SAL_AMT OF PERS_WORKSPACE = SALARY_AMOUNT
    OF SALARY_HISTORY_SCREEN
THEN
    MOVE "F" TO NOT_FOUND
ELSE
&RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
&RDB&   EMPLOYEE_ID IN JOB_HISTORY_SCREEN
&RDB&   AND SH.SALARY_END MISSING
&RDB&     ON ERROR
            PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
            &RDB&  ROLLBACK
            GO TO 030-UPDATE-INFO
&RDB&     END_ERROR

        MOVE "F" TO NOT_FOUND

&RDB&   MODIFY SH USING
&RDB&     ON ERROR
            PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
            &RDB&  ROLLBACK
            GO TO 030-UPDATE-INFO
&RDB&     END_ERROR
&RDB&       SH.SALARY_END = JOB_START IN JOB_HISTORY_SCREEN
&RDB&   END_MODIFY
&RDB& END_FOR

IF NOT_FOUND = "T"
THEN
&RDB&    ROLLBACK
         MOVE REC-NOT-FOUND TO ERROR_FIELD
         GO TO 030-UPDATE-INFO
END-IF

&RDB& STORE SH IN SALARY_HISTORY USING
&RDB&   ON ERROR
          PERFORM 060-ERROR-CHECK THRU 060-ERROR-CHECK-EXIT
          &RDB&    ROLLBACK
          GO TO 030-UPDATE-INFO
&RDB&   END_ERROR
```

Example 3-5: Update Program Using TDMS Calls (Cont.)

```
    &RDB&      SH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_SCREEN;
    &RDB&      SH.SALARY_AMOUNT = SALARY_AMOUNT IN SALARY_HISTORY_SCREEN;
    &RDB&      SH.SALARY_START = JOB_START IN JOB_HISTORY_SCREEN
    &RDB& END_STORE
    END-IF.

    &RDB& COMMIT.

    GO TO 020-RETRIEVE-INFO.

040-CLEANUP.
    &RDB& FINISH

    CALL "TSS$CLOSE_RLB" USING
         BY REFERENCE LIBRARY-ID
         GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

    CALL "TSS$CLOSE" USING
         BY REFERENCE CHANNEL,
         BY REFERENCE CLEAR-SCREEN
         GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
        CALL "TSS$SIGNAL" GIVING STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

060-ERROR-CHECK.
    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO ERROR_FIELD
    ELSE
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

060-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

Example 3-5: Update Program Using TDMS Calls (Cont.)

Transaction Processing Against a Database 4

With components of the VAX Information Architecture, and especially with VAX ACMS, you can develop, use, and control transaction processing applications that allow many terminal users to perform data entry, display, and update tasks at the same time. You divide an ACMS application into units of processing work called tasks; each task has a task definition that describes the exchange of information between the terminal user and the application, and the processing of that information against a master file or database. (This manual assumes that your data is stored in either a VAX DBMS or VAX Rdb/VMS database.) Thus, ACMS provides a way to apply structured programming concepts, such as top-down design, to the entire application.

A transaction processing application developed with the VAX Information Architecture consists of the following elements:

- A file or database in which your data is stored
- VAX TDMS forms on which terminal users enter data and on which data is displayed
- VAX TDMS requests that transfer data to workspaces where it can be retrieved for display on a form or for storage in a database
- VAX CDD record definitions for miscellaneous workspaces and one or more CDD directories to contain all the definitions needed in the application
- Procedures written in a VAX high-level language that store, retrieve, and modify data in the database
- VAX ACMS tasks that control the exchange of information between the user and the application, and the processing of that information against the database

A terminal user enters an ACMS application by way of a selection menu that lists the tasks contained in the application. When the user selects a task from the menu, ACMS uses the task definition to determine the order in which it should call requests and procedures. It uses requests to display forms on the user's terminal, collect input typed by the user, and store output retrieved from the database. It uses procedures to do the actual retrieval, storage, and modification of data in the database. ACMS uses workspaces to pass information between procedures; when ACMS calls either a request or a procedure, it must pass any workspaces used to store input and output.

An ACMS application also uses the following elements:

- One or more task groups that allow a set of tasks to share common resources
- One or more menus that list the tasks a user can select
- An application definition that describes control characteristics for the tasks in the application

You define tasks, task groups, menus, and applications with VAX ACMS's Application Definition Utility (ADU). You can write the procedures in any VAX high-level language or in VAX DATATRIEVE; however, application development is much simpler if you choose a language that supports the CDD. The other elements of an ACMS application are defined with the VAX Information Architecture components described in earlier chapters of this manual.

In many respects, application development with ACMS is very similar to application development with TDMS: both use forms, requests, and high-level language procedures to interact with a database. However, the operations you perform in an ACMS application are divided into tasks, each of which is further divided into steps. In an ACMS task, input and output operations are performed not by TDMS programming calls but by **exchange steps** that call TDMS requests; processing against the database is performed by **processing steps** that call high-level language procedures. The task definition controls the execution of exchange and processing steps.

In addition, ACMS automatically opens the request library file and the channel to the terminal when you start an application and closes them when you exit from an application, removing that responsibility from the procedures. Thus, ACMS handles all the work that would be done with TDMS programming calls in a TDMS application.

The task definition also specifies what action is to be taken if the user presses a program request key or if an error occurs during processing. The high-level language procedures in an ACMS application, then, contain little more than the data manipulation statements needed to perform the desired operations on the database.

Tasks developed with ACMS can be run in a distributed environment. An ACMS terminal user on one node of a network (that is, a VAXcluster, a local area network, or a wide area network) can select and run tasks on other nodes of the network. The user does not have to set host to the other node, and the task definition does not have to be changed; the distribution is handled transparently by ACMS. The *VAX ACMS Application Management Guide* describes how to distribute ACMS applications on a network.

This chapter concentrates on the definitions of two ACMS tasks, one from the AVERTZ Company's personnel application, which uses the Rdb/VMS database described in Section 2.1, and one from AVERTZ's car rental application, which uses the DBMS database described in Section 2.2. These tasks are taken from two much larger ACMS applications, the sources for which are in Appendix A.

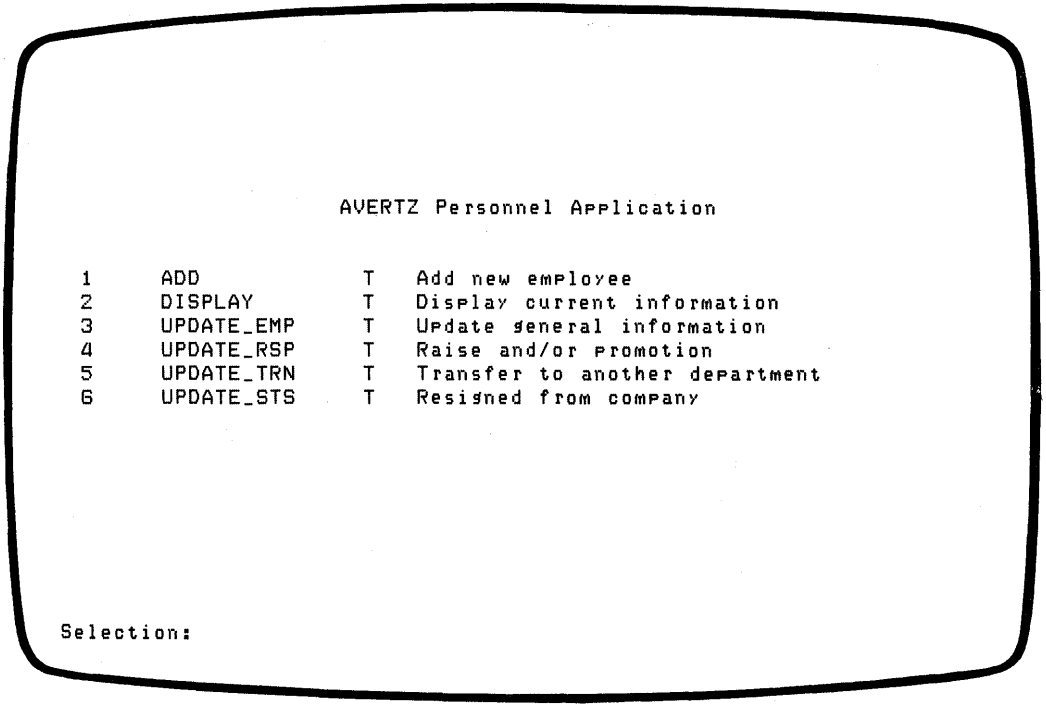
4.1 An ACMS Personnel Application

The AVERTZ Company uses an ACMS application to add new employees, display current information about an employee, and update certain employee information in the Rdb/VMS personnel database. When a user enters the personnel application, the menu shown in Figure 4-1 appears on the user's terminal screen.

When the user selects one of the tasks listed on the menu, ACMS consults the task definition and executes the task as the definition directs. The ADD task stores new records in the database, and the DISPLAY task retrieves information from the database. Each of the four update tasks involves combinations of data retrieval, storage, and update operations:

1. The user supplies the employee number for the employee whose personnel information needs to be changed.
2. The personnel information for the corresponding employee is retrieved from the database.
3. The information is displayed on the user's terminal screen, and the user can change it as necessary.
4. The new information is used to modify existing data or store new records in the database.

As explained in Chapter 3, this series of operations constitutes an inquiry/update task. Chapter 3 showed the development of such a task as a TDMS application. The following sections of this chapter show how the raise/promotion task could be implemented as part of an ACMS application.



ZK-00049-00

Figure 4-1: Personnel Application Menu

4.1.1 Defining an Inquiry/Update Task

A task definition is composed of three kinds of steps:

- Exchange steps, which call TDMS requests and handle program request keys
- Processing steps, which start database transactions, call high-level language procedures, and handle processing errors
- Block steps, which group exchange and processing steps into a unit

In an inquiry/update task, you need one exchange step to prompt the user for a key value and another to display the requested record. You need one processing step to retrieve the record from the database and another to write the record back to the database with the user's changes.

The easiest way to define a task is to create a file of ADU commands with a text editor, such as EDT, and then submit it as a command file to ADU, which checks for possible errors. The *VAX ACMS Application Definition Reference Manual* describes ADU in detail. For more information on defining tasks, see the *VAX ACMS Task Definition Guide.*

4.1.1.1 Exchange Steps -- The first exchange step of the inquiry/update task calls a TDMS request to display a form. This form prompts the user to type a value for the key field (the employee number), which the request then transfers to a workspace. A procedure can use this value to retrieve an employee record from the database. The TDMS request in the second exchange step displays the retrieved record on the same form. After the user changes the contents of the form fields, the request stores the modified record in a workspace.

If an error occurs during a processing step, you need to repeat the previous exchange step to display the form with the data that the user entered and an error message describing the error. The user can then decide whether to try the operation again or exit from the task. Both exchange steps should allow the user to press a program request key to exit easily from the task.

The two requests used in this task are very similar to the requests used in the TDMS application in Chapter 3 and use the same form. Sections A.1.6.3 and A.1.6.4 show the complete request definitions used in the inquiry/update task.

In an ACMS application, much of a task's error handling can be done in the task definition, using a special workspace called ACMS$PROCESSING_STATUS. The four fields in this workspace contain the following information:

- The status value returned by a procedure
- The severity level of the status (success, information, warning, error, or fatal)
- The status type (good or bad)
- The error message obtained from a message file, if you choose to use one

After a procedure executes, the task definition can check the status value stored in the workspace. If an error occured, ACMS can get the corresponding message from the message file and call a request to display the message on the user's terminal. If you want to use ACMS$PROCESSING_STATUS for error handling, you must pass this workspace to the request. Thus, the requests in the

inquiry/update task must pass ACMS$PROCESSING_STATUS, but they do not need a separate workspace field for storing status results. Because the requests do need to store other miscellaneous information, such as the value of a program request key, they use PERS_WORKSPACE, just as the TDMS application does. And finally, the error messages are not included directly in the requests but are stored in a message file, as described in Section 4.3.2.

To write an exchange step in a task definition, you begin with the ADU keyword EXCHANGE. You then use the REQUEST clause to specify the name of the TDMS request called in that step and list the workspaces it uses. You also need to test the value of the field that holds the program request key to see whether the user pressed the program request key when the request was called; if so, you must direct ACMS to take the appropriate action. You use a CONTROL FIELD clause in the task definition to test the field's contents and an EXIT TASK clause to direct ACMS to exit from the task if the request stored the value "EXIT" in the field.

The exchange steps for the inquiry/update task are:

```
EXCHANGE
   REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST1
      USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE;
   CONTROL FIELD IS PROGRAM_REQUEST_KEY
      "EXIT" : EXIT TASK;
   END CONTROL FIELD;


EXCHANGE
   REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST2
      USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE,
      SALARY_HISTORY;
   CONTROL FIELD IS PROGRAM_REQUEST_KEY
      "EXIT" : EXIT TASK;
   END CONTROL FIELD;
```

4.1.1.2 Processing Steps -- The first processing step of an inquiry/update task calls the COBOL procedure that retrieves the requested record from the database; the second processing step calls the COBOL procedure that writes the modified record back to the database. You begin each processing step with the ADU keyword PROCESSING. You then use a CALL clause to name the procedure, the procedure server in which it runs, and the workspaces passed to it. If you write your procedure in VAX COBOL, the name you use in the CALL clause is the PROGRAM-ID.

When ACMS starts a processing step, it allocates a server process to handle the procedure in that step. A server process is a specialized VMS process with a user name, privileges, and quotas, just like your own VMS process. Each server process has a definition that specifies the characteristics it needs to run the procedures. More than one server process can be created from the same definition,

and more than one procedure can use a single server. The server allows the procedures to execute more efficiently by performing common work for them when the server is started, rather than each time a task is selected. For example, a server can save the system overhead involved in file processing by opening files when a server is started and closing them when the server is stopped, rather than opening and closing files for each task.

When you call a procedure, you start an ACMS recovery unit that corresponds to a DBMS or Rdb/VMS transaction. When you are writing an Rdb/VMS application, you use the phrase WITH RDB RECOVERY and an RDO START_TRANSACTION statement to indicate the type of operation you intend to perform, the relations you will use, and the extent to which other users may or may not access those relations. This START_TRANSACTION statement is no different in form than the statement issued in the TDMS application in Chapter 3 or in the interactive use of RDO shown in Chapter 2.

The processing steps should include some means for detecting and reporting errors. There are two common errors in a retrieval and update operation:

- The user might type a key value that does not exist in any of the records in the database.
- The record requested by the user might be locked by another user who is attempting to update it.

The user can recover from either of these errors by reentering the key value or by retrying the operation. A more serious error occurs, however, if the database has been corrupted in some way. If that happens, the user cannot correct the error but should notify the database administrator.

Note that these procedures do not handle one situation that may arise in an inquiry/update operation: when a user tries to replace a record in the database, the record may have been modified since the user first saw it. In that case, the user is attempting to modify an outdated version of the record. If this situation seems likely to occur frequently in your application, you can handle it in one of several ways. For example, you can store a copy of the record when the user first retrieves it. Then, before the user's changes are written to the database, you can compare the saved copy with the current version of the record. If the two records do not match, you can notify the user of the discrepancy and display the current record.

With the DML statements that retrieve a record, your procedure can use an ON ERROR clause to test for the locked-record error and for database corruption. You cannot detect the nonexistent-record error with an ON ERROR clause, however; you must include statements in your procedure to determine whether a record was actually retrieved from the database. If an error occurs, your procedure should move an appropriate error value to the procedure return status.

ACMS stores the status result and status type of any procedure call in fields of the ACMS$PROCESSING_STATUS workspace. When control returns to the task definition, ACMS can test either field and, if an error occurred, retrieve the appropriate error message from the message file. It can then recall the retrieval request and display the message on the screen.

You use a CONTROL FIELD clause to test the return status or status type and specify the action to be taken. In this case, you need to test only the status type, because you want to take the same action for either expected error. If an error occurs, you want to:

- Obtain the appropriate error message from the message file, using the GET MESSAGE clause
- Roll back any changes that were made to the database before the error occurred, using the ROLLBACK clause
- Repeat the previous exchange step, using the GOTO PREVIOUS EXCHANGE clause, so that the message can be displayed

The user can then decide to retry the operation or exit from the task by pressing the program request key.

The processing steps for the inquiry/update task are:

```
PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
   CALL PERS_GET_RAISEPRO IN PERS_SERVER
      USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : GET ERROR MESSAGE;
            ROLLBACK;
            GOTO PREVIOUS EXCHANGE;
   END CONTROL FIELD;


PROCESSING WITH RDB RECOVERY
   "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
   "SALARY_HISTORY FOR SHARED WRITE"
   CALL PERS_UPDATE_RAISEPRO IN PERS_SERVER
      USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : GET ERROR MESSAGE;
            ROLLBACK;
            GOTO PREVIOUS EXCHANGE;
    END CONTROL FIELD;
```

4.1.1.3 Completing the Task Definition -- Once you have defined the two exchange steps and two processing steps, you can complete the inquiry/update task definition by defining the block step and listing the characteristics common to all the steps. The block step consists not only of the exchange and processing

steps, which constitute the block work, but can also include optional block attributes and actions. To define a simple block step, all you need to do is precede the first step in the task definition with the keywords BLOCK WORK and follow the last processing step with the keywords END BLOCK WORK.

You must complete the task definition with a WORKSPACES clause that lists the CDD path names (or given names) of the workspaces used in the task. The inquiry/update task uses JOB_HISTORY, SALARY_HISTORY, and PERS_WORKSPACE to pass information between processing steps. Because of the CDD hierarchy that Rdb/VMS creates when you define a database, the definitions for Rdb/VMS records are not stored in the same CDD directory as the other requests and workspaces used in the task. Thus you should use the full CDD path name for these definitions in the WORKSPACES clause. In the body of the task definition, however, you can use the record definitions' given names. You do not need to list the ACMS$PROCESSING_STATUS workspace because it is always available to an ACMS task.

The WORKSPACES clause for the inquiry/update task definition is as follows:

```
WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;
```

You complete the task definition with the END DEFINITION keywords.

Example 4-1 shows the complete task definition for the inquiry/update task.

```
WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST1
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_RAISEPRO IN PERS_SERVER
          USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
          "B" : GET ERROR MESSAGE;
                ROLLBACK;
                GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;
```

(continued on next page)

Example 4-1: Inquiry/Update Task Definition

```
   EXCHANGE
      REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST2
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE,
         SALARY_HISTORY;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
      "SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_UPDATE_RAISEPRO IN PERS_SERVER
         USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

Example 4-1: Inquiry/Update Task Definition (Cont.)

4.1.1.4 Storing the Task Definition in the CDD -- You should store the task definition for the inquiry/update task in the CDD along with the other parts of your application. You use ADU's CREATE or REPLACE command in your task definition to direct ADU to check the source file for syntax errors and, if it finds no errors, to store the definition in the CDD. The REPLACE command works exactly like the CREATE command if the definition does not already exist in the CDD. Just in case a task definition does not compile correctly the first time you submit it to ADU, you should use the REPLACE command.

You include either command as the first line in your command file, specifying the kind of definition (in this case, task) and the definition's complete path name or given name in the CDD. For example, the following command names the inquiry/update task:

```
REPLACE TASK PERS_UPDATE_RAISEPRO_TASK
```

When ADU processes this command, it checks the rest of the command file and, if it finds no errors, creates the task definition in the CDD. If your default CDD directory is set to the directory where you want to store your definitions (in this case, CDD$TOP.RDBPERS), you can just use the given name in the REPLACE command.

To enter ADU, first define ADU as a global symbol in your login command file or at DCL command level:

```
$ ADU :== $ACMSADU
```

Then, to invoke ADU, simply type ADU. At the ADU> prompt, you can submit your command file to ADU and insert your task definition in the CDD. If you stored the inquiry/update task definition in a source file called PERS_UPDATE_RAISEPRO_TASK.COM, you would submit it to ADU as follows:

```
$ ADU
ADU>@PERS_UPDATE_RAISEPRO_TASK
```

Because .COM is the default file type for ADU command files, you do not need to specify it on the command line. If ADU detects syntax errors in your task definition, you must edit the source file and resubmit it. You must repeat these two steps until the file is processed without errors.

To exit from ADU, use the EXIT command or CTRL/Z.

4.1.2 Writing the Step Procedures

An inquiry/update task requires two step procedures, one to retrieve a record from the database and one to replace the modified record. The logic involved in the two procedures is the same as that used in the TDMS application program in Chapter 3; however, the step procedures are simpler because error handling and TDMS programming calls are removed.

ACMS step procedures written in VAX COBOL are written as subprograms. In the Identification Division, you give the subprogram a name; this name must be unique among all the procedures that run in the same server, and it must be the name you specified in the processing step of your task definition. In the Environment Division, you do not need to use an Input-Output Section if your data is stored in a database.

The Data Division contains two sections, the Working-Storage Section and the Linkage Section. In the Working-Storage Section, you name the database you want to use and define condition values to use in error handling. In the inquiry/update task, you must define the Rdb/VMS condition codes for the locking conflicts you expect to occur. If you use a message file, you must also define message symbols that correspond to the error messages you want to display on the user's screen. Finally, the procedures in this task use a status result variable to control how the procedures stop executing if an error occurs.

The Linkage Section lists the workspaces being passed from the request to the procedure. You use COPY statements to indicate which CDD record definitions correspond to the various workspaces. Be sure to list the workspaces in the Linkage Section in the same order you listed them in the CALL clause of the processing step. Because the linkage record and the database record must not have

the same name, the COPY statements must include the REPLACING clause to assign different names to the workspaces. The retrieval and update procedures also use PERS_WORKSPACE for the program request key and other miscellaneous fields. You must list PERS_WORKSPACE in the Linkage Section, but, because it is not a database record, you do not need to rename it.

In the Procedure Division header, you list the workspaces used by the procedure; be sure to list them in the Procedure Division in the same order you listed them in the task definition.

4.1.2.1 The Retrieval Procedure -- In the main section of the Procedure Division, the retrieval procedure performs the following actions:

1. Sets the STATUS-RESULT variable to success and initializes PROGRAM_REQUEST_KEY with spaces.
2. Obtains the employee number that the user typed from the JOB_HISTORY workspace where the retrieval request stored it.
3. Uses the JOB_HISTORY and SALARY_HISTORY relations in the database to find the fields to be displayed and stores these fields in the JOB_HISTORY and SALARY_HISTORY workspaces, which are passed to the update request. If the records that contain these fields do not exist or are already locked, the procedure stores an error value in STATUS-RESULT and exits.

Example 4-2 shows the complete retrieval procedure for the inquiry/update task.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_GET_RAISEPRO.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED               PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_RECNOTFD.
```

Example 4-2: Retrieval Step Procedure in COBOL

```
01 DB-FAILURE                PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK             PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT        PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL                PIC S9(9) COMP
                               VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT             PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.


MAIN SECTION.
000-MAIN-PARAGRAPH.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.

    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   JH.JOB_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JOB_CODE IN JOB_HISTORY_LINKAGE = JH.JOB_CODE;
    &RDB&       DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE = JH.DEPARTMENT_CODE;
    &RDB&       JOB_START IN JOB_HISTORY_LINKAGE = JH.JOB_START;
    &RDB&       SUPERVISOR_ID IN JOB_HISTORY_LINKAGE = JH.SUPERVISOR_ID
    &RDB&   END_GET
    &RDB& END_FOR
```

(continued on next page)

Example 4-2: Retrieval Step Procedure in COBOL (Cont.)

```
    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

    MOVE "T" TO NOT_FOUND.

    MOVE JOB_CODE OF JOB_HISTORY_LINKAGE TO JOB
        OF PERS_WORKSPACE.

    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   SH.SALARY_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE = SH.SALARY_AMOUNT
    &RDB&   END_GET
    &RDB& END_FOR

    MOVE SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE TO
        SAL_AMT OF PERS_WORKSPACE.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

Example 4-2: Retrieval Step Procedure in COBOL (Cont.)

You compile a step procedure with the Rdb/VMS precompiler for COBOL. The precompiler first checks the syntax of the DML statements in your procedure and converts these statements into equivalent calls to the database. It then invokes the COBOL compiler to check the syntax of your COBOL statements and generate object code for the procedure. The compiler generates warning messages for the use of the DATE data type in the database definition; COBOL must convert the DATE data type into an equivalent numeric string. You can ignore these messages. You should use the /DEBUG qualifier when you compile a procedure so that you can later debug it with the VAX Symbolic Debugger. See the *VAX Rdb/VMS Guide to Programming* for information on compiling COBOL procedures with embedded DML statements. See the *VAX COBOL User's Guide* for information on compiling COBOL programs and interpreting COBOL error messages.

4.1.2.2 The Update Procedure -- Except for the PROGRAM-ID, the update procedure is identical to the retrieval procedure until the main section of the Procedure Division. In the Procedure Division, the update procedure performs the following actions:

1. Sets STATUS-RESULT to success and initializes PROGRAM_REQUEST_KEY with spaces.

2. Compares the job code in the JOB_HISTORY workspace with the job code stored in PERS_WORKSPACE by the retrieval procedure. If the job code has changed, the procedure stores the current date in the JOB_END field of the employee's JOB_HISTORY record and stores a new JOB_HISTORY record for the new job code. If the record does not exist or is already locked, the procedure stores an error value in STATUS-RESULT and exits.

3. Compares the salary amount in the SALARY_HISTORY workspace with the salary amount stored in PERS_WORKSPACE by the retrieval procedure. If the salary has changed, the procedure stores the current date in the SALARY_END field of the employee's SALARY_HISTORY record and stores a new SALARY_HISTORY record for the new salary. If the record does not exist or is already locked, the procedure stores an error value in STATUS-RESULT and exits.

Example 4-3 shows the complete update procedure for the inquiry/update task.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_UPDATE_RAISEPRO.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT      PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL              PIC S9(9) COMP
                             VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT           PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

Example 4-3: Update Step Procedure in COBOL

```
IF JOB_CODE OF JOB_HISTORY_LINKAGE NOT = JOB OF PERS_WORKSPACE
THEN
&RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
&RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
&RDB&   JH.JOB_END MISSING
&RDB&     ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
            IF STATUS-RESULT NOT SUCCESS
            THEN
                GO TO 100-EXIT-PROGRAM
            END-IF
&RDB&     END_ERROR

MOVE "F" TO NOT_FOUND

&RDB&   MODIFY JH USING
&RDB&     ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
            IF STATUS-RESULT NOT SUCCESS
            THEN
              GO TO 100-EXIT-PROGRAM
            END-IF
&RDB&     END_ERROR
&RDB&       JH.JOB_END = JOB_START IN JOB_HISTORY_LINKAGE
&RDB&   END_MODIFY
&RDB& END_FOR

&RDB& STORE JH IN JOB_HISTORY USING
&RDB&   ON ERROR
          PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
          IF STATUS-RESULT NOT SUCCESS
          THEN
              GO TO 100-EXIT-PROGRAM
          END-IF
&RDB&   END_ERROR
&RDB&     JH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
&RDB&     JH.JOB_CODE = JOB_CODE IN JOB_HISTORY_LINKAGE;
&RDB&     JH.DEPARTMENT_CODE = DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE;
&RDB&     JH.JOB_START = JOB_START IN JOB_HISTORY_LINKAGE;
&RDB&     JH.SUPERVISOR_ID = SUPERVISOR_ID IN JOB_HISTORY_LINKAGE
&RDB& END_STORE
END-IF.

IF NOT_FOUND = "T"
THEN
    MOVE REC-NOT-FOUND TO STATUS-RESULT
    GO TO 100-EXIT-PROGRAM.

MOVE "T" TO NOT_FOUND.

IF SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE NOT =
    SAL_AMT OF PERS_WORKSPACE
THEN
&RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
&RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
&RDB&   SH.SALARY_END MISSING
```

(continued on next page)

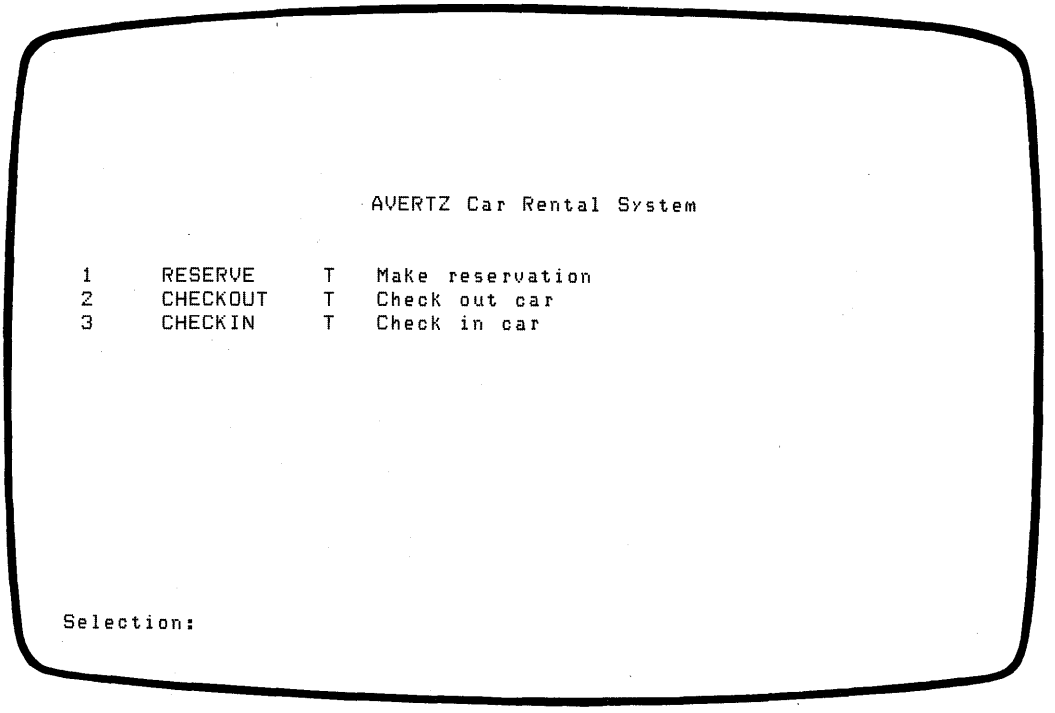
Example 4-3: Update Step Procedure in COBOL (Cont.)

```
    &RDB&      ON ERROR
                 PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                 IF STATUS-RESULT NOT SUCCESS
                 THEN
                     GO TO 100-EXIT-PROGRAM
                 END-IF
    &RDB&      END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&    MODIFY SH USING
    &RDB&      ON ERROR
                 PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                 IF STATUS-RESULT NOT SUCCESS
                 THEN
                     GO TO 100-EXIT-PROGRAM
                 END-IF
    &RDB&      END_ERROR
    &RDB&        SH.SALARY_END = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB&    END_MODIFY
    &RDB& END_FOR

    &RDB& STORE SH IN SALARY_HISTORY USING
    &RDB&    ON ERROR
               PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
    &RDB&    END_ERROR
    &RDB&      SH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
    &RDB&      SH.SALARY_AMOUNT = SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE;
    &RDB&      SH.SALARY_START = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE
    END-IF.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

Example 4-3: Update Step Procedure in COBOL (Cont.)

4.2 An ACMS Car Rental Application

The AVERTZ Company uses another ACMS application to record car rental reservations in a DBMS database, mark cars as checked out when customers arrive to pick up reserved cars, and check cars back in when customers return them. When a user enters the car rental application, the menu shown in Figure 4-2 appears on the user's terminal screen.



ZK-00050-00

Figure 4-2: Car Rental Application Menu

When the user selects one of these tasks, ACMS consults the task definition and executes the task as the definition directs. The three tasks perform the following actions:

- The RESERVE task obtains the information needed from the user to make a car reservation in the database. The user must specify a type of car to rent (compact, mid-size, or full-size) and a location at which to pick up the car.

The AVERTZ Company needs to know the customer's name, the company that the customer works for (if the customer is charging the rental to a corporate account), and the pickup date.

- The CHECKOUT task locates a customer's reservation in the database and assigns the customer a car by disconnecting a car record from the set of checked-in cars and connecting it to the customer's reservation.
- The CHECKIN task disconnects a car from a reservation and reconnects it to the set of checked-in cars. It also disconnects the customer's reservation from the pickup location to denote that the reservation has been fulfilled.

The following sections show the development of the reservation task in the car rental application.

4.2.1 Defining a Task

The reservation task consists of the following series of operations:

1. The user supplies the car type code and pickup location for the customer making a reservation.
2. The rental rates for that car type and complete address information for the requested AVERTZ location are retrieved from the database.
3. This information is displayed on the user's terminal screen, and the user is asked to supply the customer's full name, the company name (if any), and the date on which the user wants to pick up the car (which by default is the current date).
4. The new information is used to check the company's credit record (if this is a business rental), store a new customer record if the customer is not already in the database, and store the customer's reservation.
5. The user can then proceed directly to the checkout task, if the customer wants to rent the car on the current date, or exit from the task.

Like any ACMS task that requires more than one step, the reservation task consists of exchange and processing steps grouped into a block step.

4.2.1.1 Exchange Steps -- The first exchange step calls a TDMS request, AVERTZ_RESERVE_REQUEST1. This request displays a form that prompts the user to supply a car type code and pickup location. It then transfers these values to two workspaces where a procedure can use the information to retrieve car type and location records from the database. To define the workspaces, the first exchange step uses the CAR_TYPE and LOCATION record definitions, which were stored in the CDD when the DBMS database was created.

If an error occurs during a processing step, you need to repeat the previous exchange step to display an error message. The user can then decide to try the operation again or exit from the task with a program request key. The GOLD-E key combination is the program request key for this exchange step; its value is stored in a field of a miscellaneous workspace called AVERTZ_WORKSPACE. (Section A.2.2 shows the definition for AVERTZ_WORKSPACE.) In the car rental application, as in the personnel application, tasks use the ACMS$PROCESSING_STATUS workspace to handle errors and retrieve error messages from a message file.

The first exchange step of the reservation task is:

```
EXCHANGE
   REQUEST IS AVERTZ_RESERVE_REQUEST1
      USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CAR_TYPE,
      LOCATION;
   CONTROL FIELD IS PROGRAM_REQUEST_KEY
      "EXIT" : EXIT TASK;
   END CONTROL FIELD;
```

The second exchange step calls AVERTZ_RESERVE_REQUEST2 to display on a form the rental rates and location information retrieved in the previous processing step. The user is asked to supply a company name and the customer's full name. If the customer is making the reservation for a future date, the user must supply that date. The information is transferred to other workspaces that a procedure uses to store a new reservation record and possibly a new customer record.

But suppose that when the customer discovers the rates for the type of car he requested, he decides to ask for a car of a different size. The user must enter a new car type code and retrieve the corresponding rates from the database. Rather than exit from the task completely, return to the top-level menu, and reselect the reservation task, the user should be able to use a program request key to repeat the task, starting with the first exchange step. Therefore, the second request must define two program request keys, one to allow the user to exit from the task (if the customer decides to cancel the reservation, for example) and one to repeat the task from the beginning. This request defines the GOLD-E key combination to let the user exit from the task and GOLD-R to repeat it.

This exchange step uses the following workspaces:

- CAR_TYPE and LOCATION to display the information retrieved from the database
- COMPANY, CUSTOMER, and RESERVATION to store the new information supplied by the user
- AVERTZ_WORKSPACE for the program request keys
- ACMS$PROCESSING_STATUS for error handling

The following example shows the second exchange step in the reservation task:

```
EXCHANGE
   REQUEST IS AVERTZ_RESERVE_REQUEST2
      USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CAR_TYPE,
      COMPANY, CUSTOMER, LOCATION, RESERVATION;
   CONTROL FIELD IS PROGRAM_REQUEST_KEY
      "EXIT"    : EXIT TASK;
      "REPEAT" : REPEAT TASK;
   END CONTROL FIELD;
```

The third exchange step calls AVERTZ_RESERVE_REQUEST3 to redisplay the reservation form and notify the user that the reservation was successfully stored in the database. The user then has the option of proceeding directly to the checkout task by pressing the program request key. This step is designed to handle walk-in customers who reserve a car on the same day they pick it up. To inform the user of his options, the request writes a new line of text at the bottom of the form.

This request defines two program request keys, one to allow the user to exit from the task (GOLD-E) and another to indicate that the user wants to **chain** to the checkout task (GOLD-K). When you chain two or more related tasks together, the user can run the tasks in the sequence you determine when you design the application; the user does not have to return to the selection menu to choose the next task in the sequence when the previous task finishes.

This step uses the following workspaces:

- COMPANY, CUSTOMER, and RESERVATION to pass information if the user decides to chain to the next task
- AVERTZ_WORKSPACE for the program request keys
- ACMS$PROCESSING_STATUS for error handling

The following example shows the third exchange step in the reservation task:

```
EXCHANGE
   REQUEST IS AVERTZ_RESERVE_REQUEST3
      USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
      CUSTOMER, RESERVATION;
   CONTROL FIELD IS PROGRAM_REQUEST_KEY
      "EXIT"    : EXIT TASK;
      "CHKOUT" : GOTO TASK AVERTZ_CHECKOUT_TASK PASSING
                 AVERTZ_WORKSPACE, COMPANY, CUSTOMER,
                 RESERVATION;
   END CONTROL FIELD;
```

4.2.1.2 Processing Steps -- The first processing step of the reservation task calls a COBOL procedure to retrieve the rate and location information from the database. The second processing step calls a COBOL procedure to check a company's credit rating, store a customer record if the customer is not already in the database, and store a reservation record.

When you call one of these procedures, ACMS starts a recovery unit that corresponds to a DBMS transaction. You use the phrase WITH DBMS RECOVERY and a DML READY statement to indicate the type of operation you intend to perform and the extent to which other users can access the realms used in the recovery unit. This READY statement is no different in form than the statement you would issue in an interactive DBQ session such as that shown in Chapter 2.

The processing steps should include some means for detecting and reporting the following errors:

- The record requested by the user does not exist in the database (a recoverable error)
- The database is corrupted (a nonrecoverable error)

With the DML statements that retrieve records, your procedure should use the ON ERROR clause to test for these errors. The status result and status type are stored in fields of ACMS$PROCESSING_STATUS. When control returns to the task definition, ACMS can test the status type and, if an error occurred, retrieve an error message from the message file. It can then recall the previous request and display the message on the screen.

Just as in the personnel application, you use a CONTROL FIELD clause to test the status type and specify the action to be taken. If an error occurs, you want to obtain an error message, roll back any changes that were made to the database before the error occurred, and repeat the previous exchange step to display the message. The user can then decide to retry the operation or exit from the task with the program request key. The processing steps for the reservation task are:

```
PROCESSING WITH DBMS RECOVERY "READY CONCURRENT RETRIEVAL"
   CALL AVERTZ_GET_RATES IN AVERTZ_SERVER
      USING AVERTZ_WORKSPACE, CAR_TYPE, LOCATION;
   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : GET ERROR MESSAGE;
            ROLLBACK;
            GOTO PREVIOUS EXCHANGE;
   END CONTROL FIELD;


PROCESSING WITH DBMS RECOVERY "READY CONCURRENT UPDATE"
   CALL AVERTZ_RESERVE_CAR IN AVERTZ_SERVER
      USING AVERTZ_WORKSPACE, CAR_TYPE, COMPANY, CUSTOMER,
      LOCATION, RESERVATION;
   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : GET ERROR MESSAGE;
            ROLLBACK;
            GOTO PREVIOUS EXCHANGE;
   END CONTROL FIELD;
```

4.2.1.3 Completing the Task Definition -- Once you define the exchange and processing steps, you can complete the task definition by defining the block step and listing the characteristics common to all steps. You can define the block step by preceding the first step in the task with the BLOCK WORK keywords and by following the last step with END BLOCK WORK.

To complete the task definition, include a WORKSPACES clause with the CDD path names or given names of all the workspaces used in the task. As in the personnel application, you should use the complete CDD path names of the DBMS record definitions you use to define workspaces. You can use the definitions' given names in the body of the task definition.

The WORKSPACES clause for the reservation task definition is:

```
WORKSPACES ARE
   AVERTZ_WORKSPACE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;
```

You end the task definition with the END DEFINITION keywords.

Example 4-4 shows the complete task definition for the reservation task.

```
WORKSPACES ARE
   AVERTZ_WORKSPACE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;

BLOCK WORK
   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST1
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CAR_TYPE,
         LOCATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT RETRIEVAL"
      CALL AVERTZ_GET_RATES IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CAR_TYPE, LOCATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
```

Example 4-4: Reservation Task Definition

```
            ROLLBACK;
            GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST2
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CAR_TYPE,
         COMPANY, CUSTOMER, LOCATION, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
         "REPEAT" : REPEAT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT UPDATE"
      CALL AVERTZ_RESERVE_CAR IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CAR_TYPE, COMPANY, CUSTOMER,
         LOCATION, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST3
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
         CUSTOMER, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
         "CHKOUT" : GOTO TASK AVERTZ_CHECKOUT_TASK
                    PASSING AVERTZ_WORKSPACE, COMPANY, CUSTOMER,
                    RESERVATION;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

Example 4-4: Reservation Task Definition (Cont.)

4.2.1.4 Storing the Task Definition in the CDD -- You should store the task definition for the reservation task in the CDD along with the other parts of the car rental application. You use ADU's REPLACE command to direct ADU to store the definition in the CDD after checking the source file for errors. For example, the following command names the reservation task:

```
REPLACE REQUEST AVERTZ_RESERVE_TASK
```

If you stored the task definition in a source file named AVERTZ_RESERVE_TASK.COM, you can invoke ADU and submit the source file as shown here. Make sure your default CDD directory is set to the directory where you want to store your task definition (in this case, CDD$TOP.AVERTZ).

```
$ ADU
ADU>@AVERTZ_RESERVE_TASK
```

If ADU detects syntax errors in your task definition, you must edit the source file and resubmit it, repeating these two steps until the file is processed without errors.

4.2.2 Writing the Requests

The reservation task uses three requests, all of which are similar in form and content. The main purpose of the requests is to collect user input with INPUT TO instructions and display procedure output with OUTPUT TO instructions.

All the information needed in the reservation task is entered on the same form. But because there are three exchange steps in the task, the user needs to know which fields to fill for each exchange step and which program request keys are defined in each step. This information cannot be entered as background text on the form because it changes with each exchange step. Instead, you can define two fields on the form that can accept long alphanumeric strings. In each request, you can use an OUTPUT TO instruction to fill these fields with character strings that tell the user which fields require input and which program request keys are allowed. The INFORM_LINE field specifies the fields for which the request expects input, and the PRK_LINE field specifies which program request keys are defined in each request. Sections A.2.3.3, A.2.3.4, and A.2.3.5 contain the request definitions for the reservation task.

4.2.3 Writing the Step Procedures

The reservation task requires two step procedures, one to retrieve information from the database and one to store new information. In the Sub-Schema Section of each procedure, you use a DB statement to indicate the subschema you want to use and the location of the database root file. In the Working-Storage Section, you define the condition values used in error handling, including the DBMS condition code for the nonexistent-record error you expect to occur (in DBMS terms, this error occurs when DBMS detects the end of a collection). In all other respects, the principles involved in writing these procedures are similar to those used to write the inquiry and update procedures discussed in Section 4.1.2.

4.2.3.1 The AVERTZ Retrieval Procedure -- In the main section of the Procedure Division, the first procedure in the reservation task:

1. Sets the STATUS-RESULT variable to success and initializes PROGRAM_REQUEST_KEY with spaces
2. Uses the location code typed by the user and stored in the LOCATION workspace to fetch a LOCATION record
3. Moves the contents of the LOCATION record to the LOCATION workspace where they can be displayed by the next request

4. Uses the car type code supplied by the user and stored in the CAR_TYPE workspace to fetch a CAR_TYPE record

5. Moves the rate fields of the CAR_TYPE record to the CAR_TYPE workspace where they can be displayed by the next request

If either fetch operation fails to find the requested record, the procedure stores an error value in STATUS-RESULT and exits.

You compile the step procedure with DCL's COBOL command. The compiler generates warning messages for the use of the DATE data type in the subschema definition; COBOL must convert the DATE data type into an equivalent numeric string. You can ignore these messages. You should use the /DEBUG qualifier so that you can debug the procedure with the VAX Symbolic Debugger. See the *VAX COBOL User's Guide* for information on compiling COBOL programs and interpreting COBOL error messages.

Example 4-5 shows the first procedure of the reservation task in its entirety.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.         AVERTZ_GET_RATES.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.

01 LOC-NOT-FOUND             PIC S9(9) COMP
                               VALUE IS EXTERNAL AVZ_LOCNOTFD.
01 DB-FAILURE                PIC S9(9) COMP
                               VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                   PIC S9(9) COMP
                               VALUE IS EXTERNAL DBM$_END.
01 STATUS-RESULT              PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE"
     FROM DICTIONARY
     REPLACING ==CAR_TYPE. == BY ==CAR_TYPE_LINKAGE. ==.
```

(continued on next page)

Example 4-5: Retrieval Procedure in COBOL

```
COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION"
     FROM DICTIONARY
     REPLACING ==LOCATION. == BY ==LOCATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CAR_TYPE_LINKAGE
                         LOCATION_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
010-FIND-LOCATION.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Find the pickup location and move location information to the
* linkage record for display.

    MOVE LO_CODE OF LOCATION_LINKAGE TO LO_CODE OF LOCATION.

    FETCH FIRST LOCATION WITHIN LOCATION_CALC
            USING LO_CODE OF LOCATION
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE LOCATION TO LOCATION_LINKAGE.

020-FIND-CAR-TYPE.

* Find the car type and move the rental rates to the linkage
* record for display.

    MOVE CAR_TYPE_CODE OF CAR_TYPE_LINKAGE TO CAR_TYPE_CODE OF CAR_TYPE.

    FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE
            USING CAR_TYPE_CODE OF CAR_TYPE
        ON ERROR
            PERFORM 052-ERROR-CHECK THRU 052-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

030-GET-RATES.
    MOVE CAR_TYPE TO CAR_TYPE_LINKAGE.

    GO TO 100-EXIT-PROGRAM.
```

Example 4-5: Retrieval Procedure in COBOL (Cont.)

```
050-ERROR-CHECK.
* If location is not found, display a message; signal any other errors

    IF DB-CONDITION EQUAL DBM$_END
       MOVE LOC-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.

052-ERROR-CHECK.
* If car type is not found, signal (form definition prevents user from
* entering a car type other than 01, 02, or 03)

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

052-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

Example 4-5: Retrieval Procedure in COBOL (Cont.)

4.2.3.2 The AVERTZ Storage Procedure -- In the main section of the Procedure Division, the second procedure in the reservation task:

1. Sets STATUS-RESULT to success and initializes PROGRAM_REQUEST_KEY with spaces
2. If the user supplied information for the company name field, uses the company name stored in the COMPANY workspace to fetch a COMPANY record
3. If the company's credit rating is bad, or if the company does not exist in the database, stores an error value in STATUS-RESULT and exits
4. Uses the customer name typed by the user and stored in the CUSTOMER workspace to see whether information for that customer is already in the database
5. If the customer is not on file, stores a new CUSTOMER record and connects it to the current COMPANY record if there is one

6. Uses the location code typed by the user and stored in the LOCATION workspace to fetch a LOCATION record and obtain the next available reservation number

7. Uses the reservation number, the pickup location and car type code already available, and the pickup date stored in the RESERVATION workspace to store a RESERVATION record

Example 4-6 shows the second procedure of the reservation task in its entirety.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.         AVERTZ_RESERVE_CAR.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.

01 COM-NOT-FOUND              PIC S9(9) COMP
                                VALUE IS EXTERNAL AVZ_COMNOTFD.
01 CREDIT-BAD                 PIC S9(9) COMP
                                VALUE IS EXTERNAL AVZ_CREDITBD.
01 DB-FAILURE                 PIC S9(9) COMP
                                VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                   PIC S9(9) COMP
                                VALUE IS EXTERNAL DBM$_END.
01 DBM$_DUPNOTALL             PIC S9(9) COMP
                                VALUE IS EXTERNAL DBM$_DUPNOTALL.
01 STATUS-RESULT              PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE
     FROM DICTIONARY
     REPLACING ==CAR_TYPE. == BY ==CAR_TYPE_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY"
     FROM DICTIONARY
     REPLACING ==COMPANY. == BY ==COMPANY_LINKAGE  ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER
     FROM DICTIONARY
     REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION
     FROM DICTIONARY
     REPLACING ==LOCATION. == BY ==LOCATION_LINKAGE. ==.
```

Example 4-6: Storage Procedure in COBOL

```
COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION"
     FROM DICTIONARY
     REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CAR_TYPE_LINKAGE
                         COMPANY_LINKAGE
                         CUSTOMER_LINKAGE
                         LOCATION_LINKAGE
                         RESERVATION_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
010-COMPANY-CHECK.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* See whether customer is using a corporate account.  If so,
* check that the company's credit is OK.  If the credit is not OK,
* issue a message and roll back.

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        PERFORM 015-CREDIT-CHECK THRU 015-CREDIT-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM
    ELSE
        GO TO 020-CUSTOMER-CHECK.

015-CREDIT-CHECK.
    MOVE CO_NAME OF COMPANY_LINKAGE TO CO_NAME OF COMPANY.

    FETCH FIRST COMPANY WITHIN COMPANY_CALC
            USING CO_NAME OF COMPANY
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 015-CREDIT-CHECK-EXIT.

    IF CO_CREDIT_CHECK OF COMPANY = "NO"
    THEN
        MOVE CREDIT-BAD TO STATUS-RESULT.

015-CREDIT-CHECK-EXIT.
    EXIT.
```

(continued on next page)

Example 4-6: Storage Procedure in COBOL (Cont.)

```
020-CUSTOMER-CHECK.

* See whether customer is on file.  If not, add new customer
* information. If the new customer is an employee of a company
* on file, connect the customer to the company.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            PERFORM 025-NEW-CUSTOMER THRU 025-NEW-CUSTOMER-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    MOVE CUSTOMER TO CUSTOMER_LINKAGE.

    GO TO 040-STORE-RESERVATION.

025-NEW-CUSTOMER.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        PERFORM 028-ADD-CUSTOMER THRU 028-ADD-CUSTOMER-EXIT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        CONNECT CUSTOMER TO EMPLOYEE.

025-NEW-CUSTOMER-EXIT.
    EXIT.

028-ADD-CUSTOMER.
    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.
    MOVE SPACES TO CU_ADDR_DATA_1 OF CUSTOMER.
    MOVE SPACES TO CU_ADDR_DATA_2 OF CUSTOMER.
    MOVE SPACES TO CU_CITY OF CUSTOMER.
    MOVE SPACES TO CU_STATE OF CUSTOMER.
    MOVE SPACES TO CU_POSTAL_CODE OF CUSTOMER.
    MOVE SPACES TO CU_PHONE OF CUSTOMER.
    MOVE SPACES TO CU_LICENSE_NO OF CUSTOMER.
    MOVE SPACES TO CU_LICENSE_STATE OF CUSTOMER.

    STORE CUSTOMER
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

028-ADD-CUSTOMER-EXIT.
    EXIT.
```

Example 4-6: Storage Procedure in COBOL (Cont.)

```
040-STORE-RESERVATION.

* Move reservation information into the reservation record for
* display and store the reservation under the customer and under
* the requested pickup location.

    SET STATUS-RESULT TO SUCCESS.

    MOVE LO_CODE OF LOCATION_LINKAGE TO LO_CODE OF LOCATION,
        R_PICKUP_LOCATION OF RESERVATION.

    FETCH FIRST LOCATION WITHIN LOCATION_CALC USING
            LO_CODE OF LOCATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    ADD 1 TO LO_RES_NUM OF LOCATION.

    MOVE LO_RES_NUM OF LOCATION TO RESERVATION_NUM OF RESERVATION.

    MODIFY LO_RES_NUM OF LOCATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE CAR_TYPE_CODE OF CAR_TYPE_LINKAGE TO R_CAR_TYPE_CODE
        OF RESERVATION.

    MOVE R_PICKUP_DATE OF RESERVATION_LINKAGE TO R_PICKUP_DATE
        OF RESERVATION.

    STORE RESERVATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE 'Y' TO RES_MADE OF AVERTZ_WORKSPACE.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* If company not found, display an error message; signal any other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE COM-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".
```

(continued on next page)

Example 4-6: Storage Procedure in COBOL (Cont.)

```
050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

Example 4-6: Storage Procedure in COBOL (Cont.)

4.3 Defining a Task Group

The tasks of an ACMS application must be combined into one or more task groups so that they can share common resources, such as workspaces, a request library, and a procedure server. The tasks in a task group can also share initialization and termination procedures and a message file, as described in the following sections.

When you define a task group, you name the tasks that belong to the group and the server in which they run. If you use TDMS requests to get information from the terminal user, you must also list the request library file in which those requests are stored. You define a task group with commands and clauses of the Application Definition Utility (ADU). You can create the definition as a source file of ADU commands and submit it as a command file to ADU; if ADU finds no errors, it inserts the task group definition in the CDD.

You must convert the task group definition into a database of information that ACMS can use in running the application. Building the task group also creates an object module for the procedure server. Eventually you link this object module with the object modules for the step procedures to produce an executable server image. You can run this image under the control of the VAX Symbolic Debugger and the ACMS Task Debugger to test the procedures and tasks in the group. Section 4.3.3 briefly describes the ACMS Task Debugger.

For information on defining a simple task group, see *Developing Applications with VAX ACMS*. The *VAX ACMS Task Definition Guide* contains a more detailed discussion of defining and building task groups.

4.3.1 Writing Server Procedures

Because ACMS can use one server process to handle several step procedures, any startup and cleanup operations can be done just once during the lifetime of a process rather than at every processing step. For example, a database must be invoked or readied before the procedures in an application can retrieve a record, and access to it must be finished when all transactions have been processed. You can write an initialization procedure to perform startup actions and a termination procedure to perform cleanup actions for all the step procedures in the server.

In an initialization procedure, you can start a read-write transaction that makes all records or realms in the database available to the application. The procedure should also test the status of the operation and stop the server process if the database was not opened successfully.

In a termination procedure, you can finish access to the database. When you stop an application that uses the server, ACMS also stops the server process and executes the termination procedure.

The *VAX ACMS Application Programming Guide* includes a detailed discussion and examples of such procedures for applications that use DBMS and Rdb/VMS databases.

4.3.2 Using Message Files

When an error occurs during a processing step, you should display a message on the user's terminal screen to describe the error. You can obtain and display error messages in several ways, but the best way is to set up a central message file and use the fields of the ACMS$PROCESSING_STATUS workspace to store the return status, error message, and other pertinent information. You use the VMS Message Utility to define the message file, which you link with the procedure server image for a task group. The following description illustrates how the message file works with step procedures and requests to display error messages.

The personnel task illustrated in this chapter expects to get three different errors: locked record, nonexistent record, and database failure (corruption). When one of these errors occurs, the procedure stores an error value in STATUS-RESULT. These values are defined in the Data Division of the procedure to correspond to external symbols:

```
01  RECORD-LOCKED              PIC S9(9) COMP
                                 VALUE IS EXTERNAL PRS_RECLOCK.
01  REC-NOT-FOUND              PIC S9(9) COMP
                                 VALUE IS EXTERNAL PRS_RECNOTFD.
01  DB-FAILURE                 PIC S9(9) COMP
                                 VALUE IS EXTERNAL PRS_DBFAIL.
```

These external symbols are found in the message file along with the message text to be displayed:

```
RECLOCK     <Record is locked by another user; press RETURN to try again.>
RECNOTFD    <Employee not found; try another number or exit.>
DBFAIL      <Database contains invalid data. Notify administrator.>
```

(The PRS_ prefix is specified at the top of the message file along with other file characteristics.)

To record errors and obtain error messages, ACMS uses the fields of the ACMS$PROCESSING_STATUS workspace in the following way:

- ACMS$L_STATUS contains the STATUS-RESULT value returned by the step procedure. This value is either SUCCESS or a message symbol found in the message file.
- ACMS$T_SEVERITY_LEVEL contains the severity level of the return status value.
- ACMS$T_STATUS_TYPE contains either a B or a G, depending on the severity level.
- ACMS$T_STATUS_MESSAGE contains the message text that corresponds in the message file to the message symbol stored in ACMS$L_STATUS.

The message text is not stored in the ACMS$T_STATUS_MESSAGE field unless you use the GET MESSAGE clause in a task definition to direct ACMS to obtain the error message. The task definitions shown in this chapter handle errors in the following way:

```
CONTROL FIELD IS ACMS$T_STATUS_TYPE
   "B" : GET ERROR MESSAGE;
         ROLLBACK;
         GOTO PREVIOUS EXCHANGE;
END CONTROL FIELD;
```

When the previous exchange step is repeated, the request called in that step can test the status type and display the message that was last stored in ACMS$T_STATUS_MESSAGE:

```
CONTROL FIELD IS ACMS$T_STATUS_TYPE
   "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
END CONTROL FIELD;
```

For more information about setting up message files for a task group, see the *VAX ACMS Application Programming Guide.* For information on the Message Utility, see the VAX/VMS Utilities Reference Volume in the VMS documentation set.

4.3.3 Debugging the Tasks in the Task Group

Before you build an application that contains the task groups you have defined, you should test the tasks and make sure they execute properly. Under the control of the ACMS Task Debugger, you can simulate how a task will run as part of an application, even though you have not yet defined the application and its menus. You can debug only one task at a time; thus, you can test how individual tasks work but not how several tasks work together. For example, when you run a task

under the control of the task debugger, you cannot determine whether it properly handles the record-locked error that occurs when two users try to update the same record. The task debugger uses the request library file, the task group database file, the message file (if any), and the server image file to run the tasks in a task group. If a task does not run correctly under the debugger, you must determine which definitions are incorrect, edit them, and recompile or rebuild them.

As you use the task debugger to run the tasks in a server image, you can stop a task at any time to examine and change the contents of workspaces, and then continue running the task. You should make sure that the request obtains all the necessary information from the form, stores it in the correct workspaces, and passes the workspaces to the procedure. From the task debugger, you can call the VAX Symbolic Debugger to debug step procedures and make sure that they can handle any errors you decided were likely to occur.

For more information on testing and debugging tasks, see the *VAX ACMS Application Programming Guide.* For information on using the VAX Symbolic Debugger, consult the VAX/VMS Utilities Reference Volume in the VMS documentation set.

4.4 Defining the Application Environment

The control of an ACMS application is provided by the definitions of the application environment and the menus displayed to the terminal user. The definitions of an application and its menus establish an environment in which the application can run. The application definition describes the characteristics that control the tasks, servers, and application. The menu definition describes the contents of the menus displayed to users, such as menu entries and explanatory text.

After you define an application and its menus, you build an application database and a menu database that ACMS can use at run time with the request library file and the task group database. The *VAX ACMS Application Definition Guide* describes in detail how to define applications and menus and build application and menu databases.

Application development ends with the creation of the necessary databases. To make an application available to users, the system manager must authorize users and their terminals, install the application in the directory associated with the logical name ACMS$DIRECTORY, and start the application. *Developing Applications with VAX ACMS* describes a simple application installation. For a more thorough discussion of installation and ACMS system management, see the *VAX ACMS Application Management Guide.*

4.4.1 Defining the Application

You define an ACMS application to control one or more task groups, each of which contains related tasks that run in a shared server. In an application definition, you describe characteristics that control the application, the server, and the individual tasks:

- Application characteristics include logical names for a process called the application execution controller and the gathering of Audit Trail information about the application. The controller is a VMS process that ACMS sets up to perform exchange steps, handle servers, and assign servers to processing steps. The ACMS Audit Trail is a tool that gathers statistics about an active ACMS system so that you can determine how your ACMS system and its tasks and applications are being used.
- Server characteristics include the gathering of Audit Trail information for the server, how many active server processes are allowed for the application, and the server's user name.
- Task characteristics include the gathering of Audit Trail information for individual tasks and an access control list that determines which users can run which tasks.

ACMS provides defaults for most of these characteristics, but you can change them in the application definition. You define an application with ADU commands and clauses and submit the definition to ADU for error-checking. If ADU finds no errors, it inserts the application definition in the CDD. You can then build the application database with the BUILD APPLICATION command in ADU.

4.4.2 Defining Menus

You define ACMS menus to list the tasks in an application that a user can select for execution. ACMS provides a standard menu format with fields for the following items:

- A menu title
- Task names
- The type of each entry
- Explanatory text for each entry
- A field to accept the user's selection from the menu
- A field to display a possible error message

A menu entry can be the name of another menu, thus allowing you to create a hierarchy of menus for an application. To define a menu, you must provide the name of each entry and the name of the task to which it corresponds. Optional information, such as the menu title and explanatory text, helps users decide which tasks they should select. You define a menu with ADU commands and clauses and submit the definition to ADU for error-checking. If ADU finds no errors, it inserts the menu definition in the CDD. You can then build the menu database with the BUILD command in ADU.

Querying the Database 5

Online transaction processing applications handle structured tasks that are performed repeatedly during the course of business operations. These applications store and modify data in a database; you can summarize the data with online queries and hardcopy reports and graphics. The VAX Information Architecture includes a query language, VAX DATATRIEVE, which you can invoke from DCL command level or run in an ACMS application. The AVERTZ Company uses DATATRIEVE to perform queries and produce reports and graphics on the data stored in its personnel and car rental databases. This chapter shows several examples of DATATRIEVE procedures that retrieve and format data in a variety of ways.

To enter DATATRIEVE from DCL level, first define DTR32 as a global symbol in your login command file or at DCL command level:

```
$ DTR32 :== $DTR32
```

Then, to invoke DATATRIEVE, simply type DTR32. At the DTR> prompt, you can begin typing DATATRIEVE commands. For detailed information about DATATRIEVE commands, type HELP at the DTR> prompt or see the *VAX DATATRIEVE Reference Manual.*

To run DATATRIEVE in an ACMS application, you must use a DCL server to handle the processing work. You can define a task that invokes DATATRIEVE and then displays the DTR> prompt, or you can define tasks that execute DATATRIEVE procedures. For more information about running tasks in DCL servers, see the *VAX ACMS Application Definition Guide.*

5.1 Accessing the Database

To access data stored in either a DBMS or an Rdb/VMS database, DATATRIEVE must be able to locate the database definitions in the CDD. When you create an Rdb/VMS database, you specify a CDD path name, which you can use in DATATRIEVE commands to identify the database.

For a DBMS database, however, you must create a database instance for DATATRIEVE. You use the DATATRIEVE command DEFINE DATABASE to give the database a DATATRIEVE name, identify the physical location of the root file, and store the DATATRIEVE database definition in the CDD. In addition, you identify the subschema through which you want to access data and the schema to which that subschema belongs. The following example defines a DATATRIEVE instance of the car rental database:

```
DTR> SET DICTIONARY CDD$TOP.AVERTZ
DTR> DEFINE DATABASE AVERTZ_DB
DFN>   USING SUBSCHEMA AVERTZSS
DFN>   OF SCHEMA CDD$TOP.AVERTZ.AVERTZSC
DFN>   ON AVERTZ$APPL:AVERTZSC.ROO;
DTR>
```

The SET DICTIONARY command sets the default CDD directory to the correct location for the database definitions already stored in the CDD. Note that after you type the DEFINE DATABASE command and a database name, the DFN> prompt appears, indicating that the subsequent clauses are part of the definition of the database instance. You signify the end of the definition with a semicolon, after which DATATRIEVE again displays the DTR> prompt.

After you define a database, you can make it available for access by readying the records or relations in it. You use the DATATRIEVE READY command and specify the name of the database, as follows:

```
DTR> READY AVERTZ_DB
```

This command readies all the records in the car rental database, whose DATATRIEVE name is AVERTZ_DB.

You can also specify access modes and options when you ready a database, just as you do with the DBMS READY statement or the Rdb/VMS START_TRANSACTION statement. The default access for DATATRIEVE with a DBMS database is SHARED READ; the default access with an Rdb/VMS database is READ_ONLY. For queries that display but do not modify the data, the default access mode is suitable because it minimizes the contention for records in the database.

When you are finished working with a database, you end access to it with the FINISH command. You can then ready the database again, specifying other records or relations to be readied, or you can exit from DATATRIEVE with the EXIT command or CTRL/Z. When you exit, DATATRIEVE automatically finishes any records or relations that are still available for access.

5.2 Retrieving Records

To retrieve data from a database with DATATRIEVE, you use a **record selection expression** to specify the conditions that the retrieved data must meet. The group of records that satisfy these conditions is called a **record stream**. DATATRIEVE has many clauses that you can use to limit a record stream; for example, you can:

- Specify the number of records to be included (FIRST and ALL clauses)
- Combine records from different sources on matching values in a field (CROSS clause)
- Retrieve only those records with a particular value in some field (WITH clause)
- Reduce the records in a stream to unique values of fields (REDUCED TO clause)
- Sort the records (SORTED BY clause)

Suppose you want to combine employee and job history data for employees who started their present jobs after July 1, 1985. The following example shows a record selection expression that combines records from the EMPLOYEES and JOB_HISTORY relations on the common EMPLOYEE_ID field:

```
E IN EMPLOYEES CROSS
JH IN JOB_HISTORY OVER
EMPLOYEE_ID WITH
JH.JOB_START GE '01-JUL-1985' AND
JH.JOB_END MISSING SORTED BY
JH.DEPARTMENT_CODE, JH.JOB_START
```

Because this record selection expression is rather complicated, it uses **context variables** to name each record source and simplify references to them elsewhere in the expression. Thus, EMPLOYEES is abbreviated to E and JOB_HISTORY to JH. The CROSS clause combines these two relations on identical values in the EMPLOYEE_ID field. The WITH clause limits the record stream and a SORTED BY clause sorts the records by department code and job starting date.

If you include this record selection expression in a PRINT statement, DATATRIEVE displays all the fields in both relations on your terminal screen. For example:

```
DTR> PRINT E IN EMPLOYEES CROSS
CON>       JH IN JOB_HISTORY OVER
CON>       EMPLOYEE_ID WITH
CON>          JH.JOB_START GE '01-JUL-1985' AND
CON>          JH.JOB_END MISSING SORTED BY
CON>          JH.DEPARTMENT_CODE, JH.JOB_START
```

This statement creates a record stream from the records in the EMPLOYEES and JOB_HISTORY relations that satisfy the criteria in the WITH clause. The indentation simply helps you see which relations are being crossed and which fields are used in the record selection expression; it does not affect the execution of the statement.

When displaying information on the screen, DATATRIEVE arranges the fields in columns and determines the width of each column by the size of the data stored in the field. It also identifies each column by using the field name as the column header. To display all the fields in the EMPLOYEES and JOB_HISTORY relations, DATATRIEVE needs more than the standard 80 columns on your terminal screen. You can increase the size of the display by using the FN$WIDTH(132) function to change your terminal characteristics and the SET COLUMNS_PAGE = 132 command to spread the output across a wider screen.

Instead of increasing the screen display, you might decide that you can get the information you need from a subset of the fields. You can include a print list in the PRINT statement so that DATATRIEVE prints only the specified fields of each record in the record stream. For example, if you want to print only the department code, job code, job starting date, and employee name, you could add a print list to the previous PRINT statement:

```
DTR> PRINT JH.DEPARTMENT_CODE, JH.JOB_CODE,
CON>       JH.JOB_START, E.FIRST_NAME,
CON>       E.MIDDLE_INITIAL, E.LAST_NAME OF
CON>    E IN EMPLOYEES CROSS
CON>    JH IN JOB_HISTORY OVER
CON>    EMPLOYEE_ID WITH
CON>       JH.JOB_START GE '01-JUL-1985' AND
CON>       JH.JOB_END MISSING SORTED BY
CON>       JH.DEPARTMENT_CODE, JH.JOB_START
```

The output from this statement might look as follows:

```
DEPARTMENT JOB       JOB        FIRST      MIDDLE        LAST
   CODE    CODE     START       NAME       INITIAL       NAME

   ADMN    EENG  9-JUL-1985 Beverly         S.      Williams
   ADMN    PRSD 18-JUL-1985 Joseph          R.      Anderson
   MBMS    CLRK  7-JUL-1985 John            H.      O'Reilly
   MSCI    ADMN 21-JUL-1985 Charlotte       E.      Davis
   PERL    DMGR  1-JUL-1985 Stephen         J.      Sumner
   SUSA    ADMN 14-JUL-1985 Wendy           A.      Manning
```

The FOR statement is often a convenient way to access information from a DBMS database through the set relationships you defined in the database. With the FOR statement, you establish DBMS currency pointers to locate records within a set. By nesting FOR statements, you can navigate a DBMS database to find the set occurrence you want. You add the WITHIN clause to a record selection expression to specify the set to which records belong. The following example

uses nested FOR statements to display information about the cars checked in at the AVERTZ branch office in Tucson:

```
DTR> FOR LOC IN LOCATION WITH LO_CODE = 'TU'
CON>    FOR CT IN CAR_TYPE WITHIN TYPE_AVAILABLE
CON>       FOR C IN CAR WITHIN CHECKED_IN_CARS
CON>          PRINT C
```

The first FOR statement selects the LOCATION record with a location code of 'TU' as the current record, and the second FOR statement selects a current CAR_TYPE record in the TYPE_AVAILABLE set. With the last FOR statement, DATATRIEVE processes the CAR records owned by the current CAR_TYPE record in the CHECKED_IN_CARS set. The output from this statement might appear as follows:

```
                CAR
   CAR          TYPE          CAR     CAR    LICENSE      LICENSE
   NUM          CODE          MAKE    YEAR     NUM          STATE

  47477932             2 Dodge        85   5332355072      AZ
   4563498             3 Ford         85   9667972096      AZ
  80080160             3 Ford         85   5656366592      AZ
  71888048             3 Ford         85   3589714176      AZ
```

5.3 Defining Procedures

If you plan to execute a given series of DATATRIEVE commands or statements fairly often, you can save typing time and reduce the likelihood of mistakes by storing the commands and statements in a **procedure**. You create a procedure by using the DEFINE PROCEDURE command to give a name to a series of DATATRIEVE operations; DATATRIEVE stores the procedure definition in your default CDD directory.

If you need to change a procedure definition, you can issue the EDIT command, specifying the procedure name, to invoke the VAX EDT editor. When you exit from the editor, DATATRIEVE stores the new version of the definition in the CDD. See the *VAX DATATRIEVE Handbook* for more information on the DATATRIEVE Editor.

For example, you could define a procedure that displays a list of all the cars checked in at the TU location:

```
DTR> DEFINE PROCEDURE TU_CARS
DFN> FOR LOCATION WITH LO_CODE = 'TU'
DFN>   FOR CAR_TYPE WITHIN TYPE_AVAILABLE
DFN>      FOR C IN CAR WITHIN CHECKED_IN_CARS
DFN>         PRINT C
DFN> END_PROCEDURE
DTR>
```

Note that after you type the DEFINE PROCEDURE command and a procedure name, the DFN> prompt appears, indicating that the commands and statements you type will be included in the procedure definition. You use the END_PROCEDURE command to signify the end of the procedure, after which DATATRIEVE again displays the DTR> prompt.

The TU_CARS procedure, however, is limited to a very specific use because the selection criteria (WITH LO_CODE = 'TU') is specified in the record selection expression in the FOR statement. If you wanted to produce a similar list of cars checked in at other locations, you would have to define another procedure that differs from TU_CARS only in the value of the LO_CODE field. You can make a procedure more flexible by prompting the user for the selection criteria rather than specifying it in the procedure definition. DATATRIEVE then inserts the user-supplied value in the FOR statement each time it executes the procedure.

To generate a prompt with DATATRIEVE, you use a **prompting value expression** that consists of an asterisk (*), a period, and a character string enclosed in quotation marks. DATATRIEVE translates the asterisk into the word "Enter", appends the character string to it, and adds a colon to compose a full prompt. Suppose you want to define a procedure like TU_CARS that prompts the user to supply a location code. You can create a prompting value expression, putting it outside the FOR loop in an assignment statement. When DATATRIEVE executes the statement, it prompts the user for input and assigns the user-supplied data to a variable. Then you can substitute the variable name for the location code in the WITH clause.

You use the DECLARE statement in DATATRIEVE to define a variable and specify the type of data it can contain. For a variable that stores text, you use a picture string and specify the number of characters with As. For a two-character location code that can contain only alphabetic characters, the picture string is PIC AA. The following example defines a procedure called CARS_ON_HAND that prompts the user for a location code:

```
DTR> DEFINE PROCEDURE CARS_ON_HAND
DFN> DECLARE LC PIC AA.
DFN> LC = *."the location code in capital letters"
DFN> FOR LOC IN LOCATION WITH LO_CODE = LC
DFN>    FOR CT IN CAR_TYPE WITHIN TYPE_AVAILABLE
DFN>       FOR C IN CAR WITHIN CHECKED_IN_CARS
DFN>         PRINT C
DFN> END_PROCEDURE
DTR>
```

To execute a procedure, you type the procedure name, preceded by a colon (:), at the DTR> prompt. When you execute the CARS_ON_HAND procedure, DATATRIEVE displays the prompt and waits for you to supply input:

```
DTR> :CARS_ON_HAND
Enter the location code in capital letters: BA
```

DATATRIEVE uses the location code you typed to establish the current LOCATION record in the database.

To document a procedure, you can include comments either inside the procedure definition on lines beginning with an exclamation point (!) or as character strings in PRINT statements where they will be displayed on the terminal. The following example modifies the CARS_ON_HAND procedure to show both types of comments:

```
DTR> EDIT PROCEDURE CARS_ON_HAND
REDEFINE PROCEDURE CARS_ON_HAND
DECLARE LC PIC AA.
!
PRINT "This procedure prints a list of"
PRINT "the cars on hand at any location.", SKIP
!
! Prompt for location code to find current location record
!
LC = *."the location code in capital letters"
!
FOR LOC IN LOCATION WITH LO.CODE = LC
!
! Process all car types owned by current location
!
   FOR CT IN CAR_TYPE WITHIN TYPE_AVAILABLE
!
! Print all car records for every car type at current location
!
      FOR C IN CAR WITHIN CHECKED_IN_CARS
         PRINT C
!
END_PROCEDURE
```

Although including comments makes the procedure longer, the comments displayed on the screen explain to the user what the procedure does, and the embedded comments explain to a DATATRIEVE programmer how the procedure works.

You could also define a procedure that includes the personnel example shown earlier, prompting the user for a job starting date instead of including the date directly in the FOR statement. The procedure would then be able to display job information for any starting date. For example:

```
DTR> DEFINE PROCEDURE JOB_CHANGES
DFN> !
DFN> PRINT "This procedure prints information about all employees"
DFN> PRINT "who started their current jobs on or after the date"
DFN> PRINT "you specify.", SKIP
DFN> !
DFN> ! Declare variable for job starting date and prompt user for it.
DFN> !
DFN> DECLARE STARTING_DATE USAGE DATE.
DFN> STARTING_DATE = *."the job starting date"
DFN> !
DFN> ! Cross EMPLOYEES and JOB_HISTORY records and select the JOB_HISTORY
DFN> ! records for each employee's current job (indicated by a missing
DFN> ! job ending date)
```

```
DFN> !
DFN> FOR E IN EMPLOYEES CROSS
DFN>     JH IN JOB_HISTORY OVER
DFN>     EMPLOYEE_ID WITH
DFN>         JH.JOB_START GE STARTING_DATE AND
DFN>         JH.JOB_END MISSING SORTED BY
DFN>         JH.DEPARTMENT_CODE, JH.JOB_START
DFN>     PRINT JH.DEPARTMENT_CODE, JH.JOB_CODE,
DFN>           JH.JOB_START, E.EMPLOYEE_ID
DFN> END_PROCEDURE
DTR>
```

When you execute this procedure, DATATRIEVE issues the prompt and waits for you to supply a date:

```
DTR> :JOB_CHANGES
Enter the job starting date: 01-JAN-1985
```

DATATRIEVE assigns the date to the variable STARTING_DATE and inserts that value in the WITH clause that selects JOB_HISTORY records for the record stream.

5.4 Writing Reports

The previous sections have shown how DATATRIEVE can display information on the terminal in response to PRINT statements in ad hoc queries and procedures. Such displays are in fact simple reports, but DATATRIEVE can produce more elaborate reports with its Report Writer, which also has built-in statistical functions for calculating running totals, averages, minimum and maximum values, and other summary information. See the *VAX DATATRIEVE Guide to Writing Reports* for detailed information about the DATATRIEVE Report Writer.

To create a DATATRIEVE report, you combine a series of Report Writer statements in a **report specification**, which determines the format and content of a report. If you need to produce the same report periodically, you can simplify the task by defining a procedure that includes the report specification; then, to produce the report, you simply execute the procedure.

Before you can write a report specification, however, you must decide what information you need to include in your report and how you want the information to look on the screen or on paper. Suppose you want to create a procedure that is similar to the JOB_CHANGES procedure but that produces a formatted report. You want to sort the records by department code and job starting date, as before, but for the report, you want to group employees within their departments and print the department code only once. You also want to format the report attractively, with centered headings and blank lines for readability. You must keep the output format in mind as you create the report specification.

5.4.1 Creating a Record Stream for the Report

A report specification must contain a REPORT statement, which creates a record stream for the report, and an END_REPORT statement. In the REPORT statement, you include a record selection expression to identify the records or relations you need, optionally limiting the record stream with the clauses described in Section 5.2.

If you want to produce a hard copy of the report, you can use the ON clause in the record selection expression to indicate where the report is to be stored. You can include either a file specification or a prompting value expression in the REPORT statement; if you use a prompting value expression, the user is prompted to enter a file specification before the report is created. To see the report on the terminal screen, the user can type TT: at the prompt.

To define the procedure JOB_CHANGES_REPORT, your first step is to construct a REPORT statement, as follows:

```
REPORT E IN EMPLOYEES CROSS
       JH IN JOB_HISTORY OVER
       EMPLOYEE_ID WITH
          JH.JOB_START GE STARTING_DATE AND
          JH.JOB_END MISSING SORTED BY
          JH.DEPARTMENT_CODE, JH.JOB_START ON
       *."the file specification for the report"
```

The record selection expression in this REPORT statement is identical to that used in the FOR statement in the JOB_CHANGES procedure. It evaluates the STARTING_DATE variable to determine which JOB_HISTORY records to include in the record stream. Therefore, the JOB_CHANGES_REPORT procedure must prompt the user for a starting date and store the user's input in a variable before executing the REPORT statement. The REPORT statement also includes an ON clause, specifying that the report is to be stored in the file indicated by the user.

5.4.2 Formatting Detail and Summary Lines

A report specification must also include at least one output statement. DATATRIEVE has two kinds of output statements: the PRINT statement prints detail lines for each record in the record stream, and the AT statement prints summary lines. In the job changes report, you want to print the job starting date, employee number, and full name for each employee whose job has changed since a specified date. Therefore, you use a PRINT statement to list the fields you want displayed. By default, DATATRIEVE uses the field name as the column heading on a report; you can supply a heading by enclosing a character string in quotation marks and including it in parentheses following the field name. You can also use

formatting elements in the PRINT statement to position the columns on the page and leave blank lines and spaces. For example:

```
PRINT JOB_START ("Date"),
      EMPLOYEE_ID ("ID"),
      FIRST_NAME|||MIDDLE_INITIAL|||LAST_NAME ("Name")
```

This statement prints three columns of information. It prints the JOB_START field under the heading Date and the EMPLOYEE_ID field under the heading ID. The concatenation character (||) joins fields into a single text string; a triple bar (|||), as used in the last line of the procedure to join FIRST_NAME, MIDDLE_INITIAL, and LAST_NAME, replaces any trailing spaces contained in a field with a single space. The concatenated fields in this example are printed under the heading Name.

A series of sorted records that have the same value in at least one field form a control group. When you are producing a report, you can direct DATATRIEVE to stop before or after it processes each control group and perform various operations on the records in the group. For example, DATATRIEVE can count the number of records, total or average the values in a given field, print headings or summary lines, and so on. The Report Writer provides two variations of the AT statement--AT TOP for printing header lines and AT BOTTOM for printing summary lines in a report.

The REPORT statement in the previous example produces two types of control groups: one for each department code and one for each job starting date within a department code. You can use an AT TOP statement to print the department code at the top of each department control group. You can also include a column header and formatting elements. For example:

```
AT TOP OF DEPARTMENT_CODE PRINT SKIP, DEPARTMENT_CODE ("Department")
```

This statement directs DATATRIEVE to leave a blank line before each department control group and to print the DEPARTMENT_CODE field under the heading Department.

5.4.3 Defining Report Characteristics

DATATRIEVE has defaults that it can use to format the pages of a report. These defaults determine the number of columns and lines per page and the number of lines and pages per report. They also cause the current date, a page number, and column headings to be printed at the top of each page of a report. To change the default characteristics, you can include SET statements within a report specification.

DATATRIEVE does not generate a default heading for a report, but you can provide one with the SET REPORT_NAME statement. For example, to give a name to the job changes report, you could use the following SET statement:

```
SET REPORT_NAME = "New Jobs by Department"
```

5.4.4 An AVERTZ Personnel Report

Example 5-1 shows the JOB_CHANGES procedure, modified to produce a report, rather than a terminal display, of all the employees who have changed jobs since the date specified by the user.

```
DTR> EDIT PROCEDURE JOB_CHANGES
DEFINE PROCEDURE JOB_CHANGES_REPORT
!
PRINT "This procedure creates a report of all employees who started"
PRINT "their current job on or after the date you specify.", SKIP
!
! Declare variable for job starting date and prompt user for it.
!
DECLARE STARTING_DATE USAGE DATE.
STARTING_DATE = *."the job starting date"
!
! Cross EMPLOYEES and JOB_HISTORY records and select the JOB_HISTORY
! records for each employee's current job (indicated by a missing
! job ending date)
!
REPORT E IN EMPLOYEES CROSS
       JH IN JOB_HISTORY OVER
       EMPLOYEE_ID WITH
          JH.JOB_START GE STARTING_DATE AND
          JH.JOB_END MISSING SORTED BY
          JH.DEPARTMENT_CODE, JH.JOB_START ON
       *."the file specification for the report"
!
SET REPORT_NAME = "New Jobs by Department"
!
! Create a control group for each department
!
AT TOP OF DEPARTMENT_CODE PRINT SKIP,
          DEPARTMENT_CODE ("Department")
!
! Within each department, print the starting date, ID number, and
! full name of each employee
!
PRINT JOB_START ("Date"),
      EMPLOYEE_ID ("ID"),
      FIRST_NAME|||MIDDLE_INITIAL|||LAST_NAME ("Name")
!
END_REPORT
END_PROCEDURE
```

Example 5-1: Definition of Job Changes Report

Example 5-2 shows a sample report, produced by the JOB_CHANGES_REPORT procedure, of all employees who have changed jobs since June 1, 1985.

```
                          New Jobs by Department                1-Jul-198
                                                                Page 1

Department          Date              ID              Name

   ADMN
                  4-Jun-1985        00264        Sarah H McCloskey
                  5-Jun-1985        00290        Stanley K Lambert
                 28-Jun-1985        00279        Edward E Cummings
   ELEL
                 25-Jun-1985        00312        Adam T Macgregor
   ELGS
                 17-Jun-1985        00254        Caroline L Winston
   MBMS
                  7-Jun-1985        00347        Elizabeth S Rockwell
                 27-Jun-1985        00349        Julia B Carter
   MKTG
                  3-Jun-1985        00397        Noah M Caulfield
                 10-Jun-1985        00218        Barney J Marino
   SUSA
                 21-Jun-1985        00296        Sonya J Cortez
```

Example 5-2: Job Changes Report

5.4.5 An AVERTZ Car Rental Report

Besides personnel reports, the AVERTZ Company needs reports on its car rental data. For example, a manager might want to know how many cars of each type have been reserved at each location for a specified period of time. Such information shows the projected activity at the various locations and helps the manager determine how to allocate the inventory of rental cars. Using the same concepts applied to the definition of the job changes report, you could create a reservation report that lists the reservations at each location that will be fulfilled during an interval specified by the user.

To prompt the user for a date, you include a prompting value expression in an assignment statement and assign the date to a variable. The reservation report uses two variables, START and END, to store the dates that mark the report

period. You must declare these variables with DECLARE statements and specify their data types before you assign values to them. For example:

```
DECLARE START USAGE DATE.
DECLARE END USAGE DATE.

START = *."the starting date of the report period"
END = *."the ending date of the report period"
```

The record selection expression used in this report creates a record stream of LOCATION records, sorted by the value of the LO_CODE field. The following REPORT statement contains the record selection expression and a prompt for the file specification of the finished report:

```
REPORT LOC IN LOCATION SORTED BY
   LOC.LO_CODE ON
   *."the file specification for the report"
```

Unlike the job changes report, the reservation report does not print detail lines; that is, the report should not display any of the data contained in the record in the record stream. Instead, it should print the total number of reservations for each car type at each location. To do so, you can declare a variable for each car type and indicate with a COMPUTED BY clause that the value of each variable depends on the value of the R_CAR_TYPE_CODE field in a RESERVATION record. You use the COMPUTED BY clause to provide a conditional expression that describes the variable's value. For example:

```
DECLARE TYPE1_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 1 AND
    R.R_PICKUP_DATE BETWEEN START AND END.

DECLARE TYPE2_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 2 AND
    R.R_PICKUP_DATE BETWEEN START AND END.

DECLARE TYPE3_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 3 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
```

COUNT is a built-in function that counts the number of detail records that meet the specified criteria. The COUNT statements used in these variable declarations count the number of RESERVATION records of each car type in each occurrence of the LOCATION_RESERVATION set. RESERVATION records are further limited by the value of the R_PICKUP_DATE field, which must fall between the dates specified by the user.

To compute the sum of all reservation counts at a location, you can declare another variable and, in its COMPUTED BY clause, include an arithmetical expression to add the values of the three variables. For example:

```
DECLARE TYPE_TOTAL COMPUTED BY
    TYPE1_COUNTER + TYPE2_COUNTER + TYPE3_COUNTER.
```

To print the reservation counts at each location, you use a PRINT statement and specify the LO_CODE field and the four reservation variables. The following PRINT statement prints the values of these fields with the specified headers:

```
PRINT LOC.LO_CODE ("Location"),
      TYPE1_COUNTER ("Compacts"),
      TYPE2_COUNTER ("Midsize"),
      TYPE3_COUNTER ("Full-size"),
      TYPE_TOTAL ("Location Total"),
      SKIP
```

When you reach the end of the record stream, you can use the TOTAL function to compute the total number of reservations of each type and the overall total. Then you can print these results in a simple AT BOTTOM OF REPORT statement:

```
AT BOTTOM OF REPORT PRINT
      "Totals",
      TOTAL TYPE1_COUNTER,
      TOTAL TYPE2_COUNTER,
      TOTAL TYPE3_COUNTER,
      TOTAL TYPE_TOTAL
```

Another way to distinguish among multiple reservation reports is to include in the report header the dates for which the report applies. You cannot use the SET REPORT_NAME statement to display the values of variables; it can display only character strings and prompting value expressions. Instead, you can declare a variable to contain the report title and print the title in an AT TOP OF PAGE statement. When you declare this variable, you use the COMPUTED BY clause to concatenate a string of text with the values of the START and END variables. For example:

```
DECLARE TITLE COMPUTED BY
    "Projected Activity from"|||START|||"to"|||END
    EDIT_STRING X(50)
```

The EDIT_STRING clause defines the total length of the title string.

When you use an AT TOP OF PAGE statement, DATATRIEVE suppresses report and column headers unless you enable them by specifying REPORT_HEADER and COLUMN_HEADER. As shown in the job changes report, you can include formatting elements such as SKIP and COL to insert

blank lines and position a report header on the page. The following AT TOP statment formats the header of the reservation report:

```
AT TOP OF PAGE PRINT REPORT_HEADER, SKIP,
                     COL 10, TITLE, SKIP,
                     COLUMN_HEADER, SKIP
```

Example 5-3 shows the complete definition of CURRENT_RES_REPORT, which produces a reservation report such as that shown in Example 5-4.

```
DEFINE PROCEDURE CURRENT_RES_REPORT
!
PRINT "This procedure produces a report of the projected number of"
PRINT "car reservations of each type at each branch during the"
PRINT "period you specify.", SKIP
!
! Declare variables for starting and ending dates of report period
!
DECLARE START USAGE DATE.
DECLARE END USAGE DATE.
!
! Declare variables to count number of reservations of each type in
! each occurrence of LOCATION_RESERVATION with pickup dates in the
! report period
!
DECLARE TYPE1_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 1 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
DECLARE TYPE2_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 2 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
DECLARE TYPE3_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 3 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
DECLARE TYPE_TOTAL COMPUTED BY
    TYPE1_COUNTER + TYPE2_COUNTER + TYPE3_COUNTER.
!
DECLARE TITLE COMPUTED BY
    "Projected Activity from"|||START|||"to"|||END
    EDIT_STRING X(50).
!
! Prompt user for the report period
!
START = *."the starting date of the report period"
END = *."the ending date of the report period"
!
```

(continued on next page)

Example 5-3: Definition of Reservation Report

```
REPORT LOC IN LOCATION SORTED BY
       LOC.LO_CODE ON
       *."the file specification for the report"
!
AT TOP OF PAGE PRINT REPORT_HEADER, SKIP,
                     COL 10, TITLE, SKIP,
                     COLUMN_HEADER, SKIP
!
! Print location code, number of reservations of each type, and
! total number at each location
!
PRINT LOC.LO_CODE ("Location"),
      TYPE1_COUNTER ("Compacts"),
      TYPE2_COUNTER ("Midsize"),
      TYPE3_COUNTER ("Full-size"),
      TYPE_TOTAL ("Location Total"), SKIP
!
! Total the reservations of each type for all locations and compute
! overall total
!
AT BOTTOM OF REPORT PRINT
     "Totals",
     TOTAL TYPE1_COUNTER,
     TOTAL TYPE2_COUNTER,
     TOTAL TYPE3_COUNTER,
     TOTAL TYPE_TOTAL
!
END_REPORT
END_PROCEDURE
```

Example 5-3: Definition of Reservation Report (Cont.)

```
                                                          20-Jun-1
                                                          Page 1

          Projected Activity from  1-Jul-1986 to 15-Jul-1986

   Location        Compacts        Midsize        Full-size        Location To

      BA               22             19               10                51

      DA               36             27               15                78

      FC               28             21               16                65

      GR               16              9               18                43

      RU               12             11                5                28

      TU               19             25               13                57

    Totals            133            112               77               322
```

Example 5-4: Reservation Report

5.5 Generating Graphics

For some purposes, it may be more appropriate to display information in a chart or graph rather than in a report. DATATRIEVE's graphics features let you easily construct useful charts and graphs from the data stored in a database. To use DATATRIEVE graphics, you must have the appropriate hardware, as described in the *VAX DATATRIEVE Guide to Using Graphics.*

Before you can create graphic displays, you must issue the SET PLOTS command to indicate where in the CDD DATATRIEVE's graphics functions are stored. By default, they are stored in the CDD$TOP.DTR$LIB.PLOTS directory when you install DATATRIEVE. For example:

```
DTR> SET PLOTS CDD$TOP.DTR$LIB.PLOTS
```

You can then use the PLOT command to specify a type of graphic display. DATATRIEVE can generate bar charts, line charts, pie charts, and scatter graphs, and can format them in a variety of ways (shaded, cross-hatched, and so forth). In the PLOT command, you also include a record selection expression to create the record stream you want to display on the chart or graph.

Using the record selection expression from the JOB_CHANGES procedure in Example 5-1, you can define a procedure to create a pie chart that shows the percentage by department of employees who have changed jobs. Example 5-5 shows the definition of such a procedure.

```
DTR> EDIT PROCEDURE JOB_CHANGES
DEFINE PROCEDURE JOB_CHANGES_PIE
!
PRINT "This procedure creates a pie chart that shows the percentage"
PRINT "by department of all employees who started their current job"
PRINT "on or after the date you specify.", SKIP
!
! Declare variable for job starting date and prompt user for it.
!
DECLARE START USAGE DATE.
START = *."the job starting date"
!
! Cross EMPLOYEES and JOB_HISTORY records and select the JOB_HISTORY
! records for each employee's current job (indicated by a missing
! job ending date)
!
PLOT PIE DEPARTMENT_CODE OF E IN EMPLOYEES CROSS
     JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
         JH.JOB_START GE START AND
         JH.JOB_END MISSING
END_PROCEDURE
```

Example 5-5: Procedure Definition for Job Changes Pie Chart

Like the JOB_CHANGES procedure, this procedure prompts the user to enter a job starting date. It then creates a pie chart on the terminal screen that shows how the total number of job changes breaks down by department. The PLOT

command specifies that the DEPARTMENT_CODE field is being plotted, so DATATRIEVE labels each segment of the pie with the corresponding department code. Figure 5-1 shows a pie chart that might result from executing this procedure.

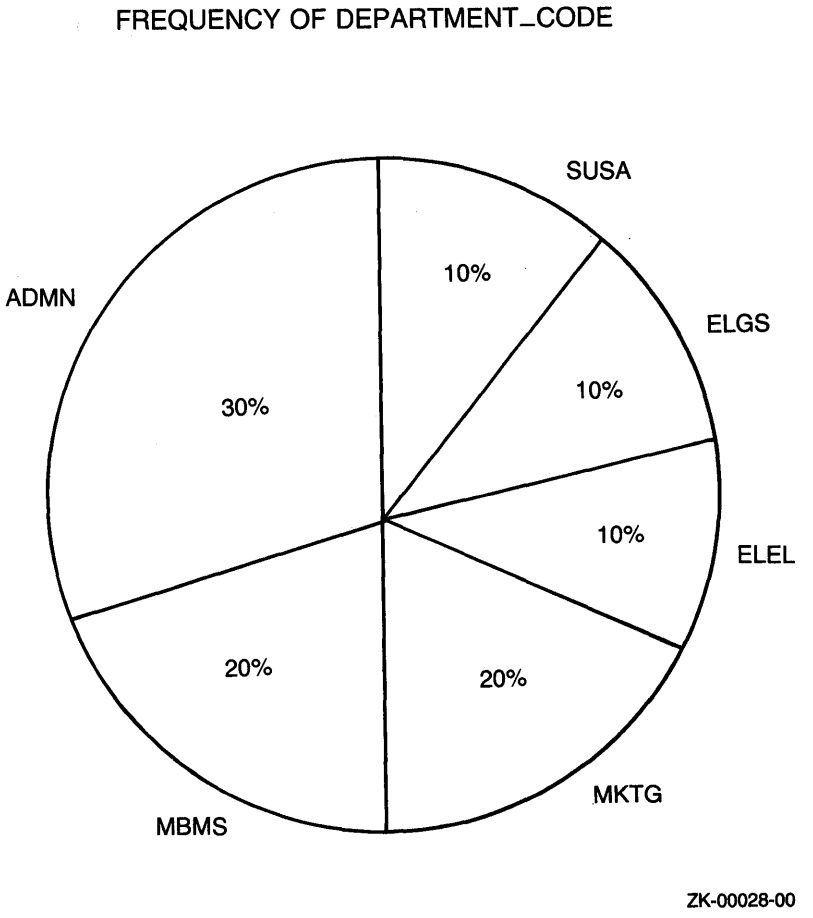


Figure 5-1: Pie Chart of Job Changes

Likewise, you could modify the record selection expression in the CURRENT_RES_REPORT procedure in Example 5-3 to create a multiple-bar chart showing the number of cars reserved at each location during a specified period. When you specify a multiple-bar chart in a PLOT statement, you must include a record selection expression to create a record stream and specify which fields of the records you want graphed. DATATRIEVE uses the first field you specify to label the bars; the remaining fields specify the data to be represented as vertical bars. To enhance the display and make the distinction between bars

easier, you can include a PLOT CROSS_HATCH statement. If there is room on the chart, DATATRIEVE also includes a legend to explain which bars represent which values.

Example 5-6 shows a procedure that conveys the same information as the report in Example 5-3.

```
DTR> EDIT PROCEDURE CURRENT_RES_REPORT
DEFINE PROCEDURE CURRENT_RES_CHART
!
PRINT "This procedure produces a bar chart of the projected number of"
PRINT "car reservations of each type at each branch during the period"
PRINT "you specify.", SKIP
!
! Declare variables for starting and ending dates of the report period
!
DECLARE START USAGE DATE.
DECLARE END USAGE DATE.
!
! Declare variables to count the number of reservations of each type
! in each occurrence of LOCATION_RESERVATION with pickup dates in the
! report period
!
DECLARE TYPE1_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 1 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
DECLARE TYPE2_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 2 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
DECLARE TYPE3_COUNTER COMPUTED BY
    COUNT OF R IN RESERVATION MEMBER LOCATION_RESERVATION WITH
    R.R_CAR_TYPE_CODE = 3 AND
    R.R_PICKUP_DATE BETWEEN START AND END.
!
! Prompt user for the report period
!
START = *."the starting date of the report period"
END = *."the ending date of the report period"
!
! Graph location code and number of reservations of each type at
! each location
!
PLOT MULTI_BAR LOC.LO_CODE ("Location"),
    TYPE1_COUNTER ("Compacts"),
    TYPE2_COUNTER ("Mid-size"),
    TYPE3_COUNTER ("Fullsize") OF
    LOC IN LOCATION SORTED BY LOC.LO_CODE THEN
    PLOT CROSS_HATCH
!
END_PROCEDURE
```

Example 5-6: Procedure Definition for Reservation Bar Chart

This procedure first prompts the user for starting and ending dates. The PLOT statement specifies LO_CODE as the first field; therefore, the location codes are

printed at the bottom of each group of bars. These vertical bars represent the number of reservations of each type made at each location. Figure 5-2 shows a chart that might result from executing this procedure.

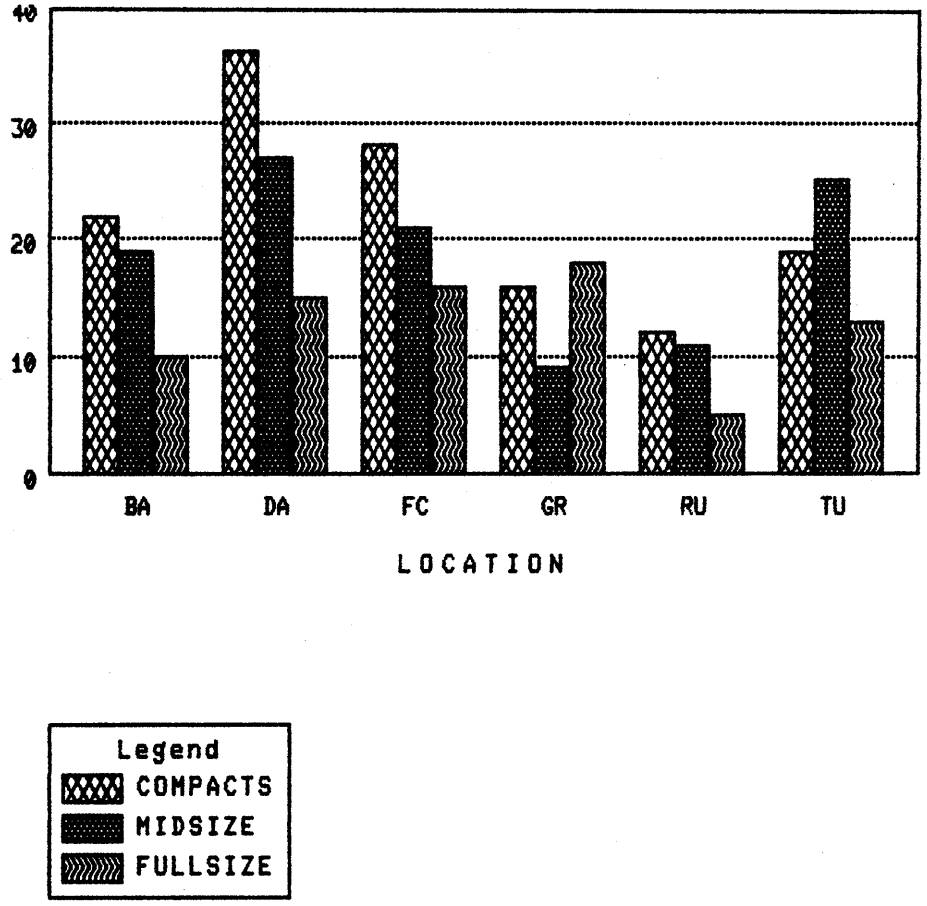


Figure 5-2: Bar Chart of Reservations for Each Location

Sources for Sample Applications A

This appendix contains the complete sources for the sample applications developed in this manual. Section A.1 includes the sources for the ACMS personnel application, and Section A.2 includes the sources for the ACMS car rental application.

A.1 AVERTZ Personnel Application

The personnel application is built on a VAX Rdb/VMS database and includes six tasks. This section contains the complete sources for the application. Table A-1 lists each type of source definition, the sections of this appendix that contain them, and the specific task to which each definition applies.

Table A-1: Personnel Application Sources

Object	Section	Related Task
Database	A.1.1	All
Workspace	A.1.2	All
Task Definitions	A.1.3.1	Add Task
	A.1.4.1	Display Task
	A.1.5.1	General Update Task
	A.1.6.1	Raise/Promotion Update Task
	A.1.7.1	Transfer Update Task

(continued on next page)

Table A-1: Personnel Application Sources (Cont.)

Object	Section	Related Task
Task Definitions	A.1.8.1	Status Update Task
Form Definitions	A.1.3.2	Add Task
	A.1.4.2	Display Task
	A.1.5.2	General Update Task
	A.1.6.2	Raise/Promotion Update Task
	A.1.7.2	Transfer Update Task
	A.1.8.2	Status Update Task
Request Definitions	A.1.3.3	Add Task
	A.1.4.3 A.1.4.4	Display Task
	A.1.5.3 A.1.5.4	General Update Task
	A.1.6.3 A.1.6.4	Raise/Promotion Update Task
	A.1.7.3 A.1.7.4	Transfer Update Task
	A.1.8.3 A.1.8.4	Status Update Task
Step Procedures	A.1.3.4	Add Task
	A.1.4.5	Display Task
	A.1.5.5 A.1.5.6	General Update Task

(continued on next page)

Table A-1: Personnel Application Sources (Cont.)

Object	Section	Related Task
Step Procedures	A.1.6.5 A.1.6.6	Raise/Promotion Update Task
	A.1.7.5 A.1.7.6	Transfer Update Task
	A.1.8.5 A.1.8.6	Status Update Task
Server Procedures	A.1.9.1 A.1.9.2	All
Request Library Definition	A.1.10	All
Task Group Definition	A.1.11	All
Message Source File	A.1.12	All
Application Definition	A.1.13	All
Menu Definition	A.1.14	All

A.1.1 Personnel Database Definition

```
! PERSONNEL database definitions
!
!
! *** Define fields for the PERSONNEL database ***
!
!
DEFINE FIELD  ID_NUMBER
   DESCRIPTION IS /* Generic employee ID */
   DATATYPE IS TEXT SIZE IS 5.
!
!
DEFINE FIELD  LAST_NAME
   DESCRIPTION IS /* Generic last name */
   DATATYPE IS TEXT SIZE IS 14.
!
!
DEFINE FIELD  FIRST_NAME
   DESCRIPTION IS/* Generic first name */
   DATATYPE IS TEXT SIZE IS 10.
!
!
```

(continued on next page)

```
DEFINE FIELD  MIDDLE_INITIAL
   DESCRIPTION IS /* Generic middle initial */
   DATATYPE IS TEXT SIZE IS 1
   EDIT_STRING FOR DATATRIEVE IS 'X.'
   MISSING_VALUE IS ' '.
!
!
DEFINE FIELD  ADDRESS_DATA_1
   DESCRIPTION IS /* Mail stops, suite addresses, street numbers, etc.*/
   DATATYPE IS TEXT SIZE IS 25
   MISSING_VALUE IS '                         '.
!
!
DEFINE FIELD  ADDRESS_DATA_2
   DESCRIPTION IS /* Street name */
   DATATYPE IS TEXT SIZE IS 25
   MISSING_VALUE IS '                         '.
!
!
DEFINE FIELD  CITY
   DESCRIPTION IS /* City name */
   DATATYPE IS TEXT SIZE IS 20
   MISSING_VALUE IS '                    '.
!
!
DEFINE FIELD  STATE
   DESCRIPTION IS /* State abbreviation (or DISTRICT) */
   DATATYPE IS TEXT SIZE IS 2
   MISSING_VALUE IS '  '.
!
!
DEFINE FIELD  POSTAL_CODE
   DESCRIPTION IS /* Postal code (in US = ZIP)*/
   DATATYPE IS TEXT SIZE IS 9
   MISSING_VALUE IS '         '.
!
!
DEFINE FIELD  SEX
   DESCRIPTION IS /* M, F */
   DATATYPE IS TEXT SIZE IS 1
   MISSING_VALUE IS '?'
   VALID IF SEX = 'M' OR
            SEX = 'F' OR
            SEX MISSING.
!
!
DEFINE FIELD  STANDARD_DATE
   DESCRIPTION IS /* Generic date field */
   DATATYPE IS DATE
   MISSING_VALUE IS '17-NOV-1858 00:00:00.00'
   EDIT_STRING FOR DATATRIEVE IS 'DD-MMM-YYYY'.
!
!
```

```
DEFINE FIELD  SALARY
   DESCRIPTION IS /* Generic salary field */
   DATATYPE IS SIGNED LONGWORD SCALE -2
   VALID IF SALARY > 0 OR
            SALARY MISSING
   EDIT_STRING FOR DATATRIEVE IS '$$$$,$$9.99'.
!
!
DEFINE FIELD  RESUME
   DESCRIPTION IS /* Employee resume */
   DATATYPE IS SEGMENTED STRING.
!
!
DEFINE FIELD  DEPARTMENT_CODE
   DESCRIPTION IS /* Department code or abbreviation */
   DATATYPE IS TEXT 4
   MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  JOB_CODE
   DESCRIPTION IS /* Generic job code */
   DATATYPE IS TEXT SIZE IS 4
   MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  WAGE_CLASS
   DESCRIPTION IS /* Wage class -- 1 to 4 */
   DATATYPE IS TEXT SIZE IS 1
   VALID IF WAGE_CLASS = '1' OR
            WAGE_CLASS = '2' OR
            WAGE_CLASS = '3' OR
            WAGE_CLASS = '4' OR
            WAGE_CLASS MISSING.
!
!
DEFINE FIELD  JOB_TITLE
   DESCRIPTION IS /* Generic job title */
   DATATYPE IS TEXT SIZE IS 20
   MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  DEPARTMENT_NAME
   DESCRIPTION IS /* Department name */
   DATATYPE IS TEXT SIZE IS 30
   MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  BUDGET
   DESCRIPTION IS /* Generic budget data */
   DATATYPE IS SIGNED LONGWORD SCALE 0
   EDIT_STRING FOR DATATRIEVE IS '$$$,$$$,$$$'.
!
!
```

(continued on next page)

```
DEFINE FIELD  COLLEGE_NAME
   DESCRIPTION IS /* Halls of ivy */
   DATATYPE IS TEXT SIZE IS 25.
!
!
DEFINE FIELD  COLLEGE_CODE
   DESCRIPTION IS /* Four-letter college code */
   DATATYPE IS TEXT SIZE IS 4.
!
!
DEFINE FIELD  YEAR_GIVEN
   DESCRIPTION IS /* Year degree awarded */
   DATATYPE IS SIGNED WORD.
!
!
DEFINE FIELD  DEGREE
   DESCRIPTION IS /* Degree awarded */
   DATATYPE IS TEXT SIZE IS 3
   VALID IF DEGREE = 'BA ' OR
            DEGREE = 'BS ' OR
            DEGREE = 'MA ' OR
            DEGREE = 'MS ' OR
            DEGREE = 'PhD' OR
            DEGREE MISSING.
!
!
DEFINE FIELD  DEGREE_FIELD
   DESCRIPTION IS /* Field in which degree was awarded */
   DATATYPE IS TEXT SIZE IS 15
   MISSING_VALUE IS 'Unknown'.
!
!
DEFINE FIELD  STATUS_CODE
   DESCRIPTION IS /* A number */
   DATATYPE IS TEXT SIZE IS 1
   MISSING_VALUE IS 'N'
   VALID IF STATUS_CODE = '0' OR
            STATUS_CODE = '1' OR
            STATUS_CODE = '2' OR
            STATUS_CODE MISSING.
!
!
DEFINE FIELD  STATUS_NAME
   DESCRIPTION IS /* Active or inactive */
   DATATYPE IS TEXT SIZE IS 8
   VALID IF STATUS_NAME = 'ACTIVE' OR
            STATUS_NAME = 'INACTIVE' OR
            STATUS_NAME MISSING.
!
!
DEFINE FIELD  STATUS_TYPE
   DESCRIPTION IS /* Full-time, part-time, or expired */
   DATATYPE IS TEXT SIZE IS 14
   VALID IF STATUS_TYPE = 'RECORD EXPIRED' OR
            STATUS_TYPE = 'FULL TIME' OR
            STATUS_TYPE = 'PART TIME' OR
            STATUS_TYPE MISSING.
```

```
!
COMMIT
!
!*********************************************************
!
!
! *** Define Relations ***
!
! START_TRANSACTION READ_WRITE
!
DEFINE RELATION EMPLOYEES.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   LAST_NAME.
   FIRST_NAME.
   MIDDLE_INITIAL.
   ADDRESS_DATA_1.
   ADDRESS_DATA_2.
   CITY.
   STATE.
   POSTAL_CODE.
   SEX.
   BIRTHDAY
      BASED ON STANDARD_DATE.
   STATUS_CODE.
END EMPLOYEES RELATION.
!
!
! Job_History Relation:
!
DEFINE RELATION JOB_HISTORY.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   JOB_CODE.
   JOB_START
      BASED ON STANDARD_DATE.
   JOB_END
      BASED ON STANDARD_DATE.
   DEPARTMENT_CODE.
   SUPERVISOR_ID
      BASED ON ID_NUMBER.
END JOB_HISTORY RELATION.
!
!
! Salary_History Relation:
!
DEFINE RELATION SALARY_HISTORY.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   SALARY_AMOUNT
      BASED ON SALARY.
   SALARY_START
      BASED ON STANDARD_DATE.
   SALARY_END
      BASED ON STANDARD_DATE.
END SALARY_HISTORY RELATION.
```

(continued on next page)

```
!
!
! Jobs Relation:
!
DEFINE RELATION JOBS.
    JOB_CODE.
    WAGE_CLASS.
    JOB_TITLE.
    MINIMUM_SALARY
        BASED ON SALARY.
    MAXIMUM_SALARY
        BASED ON SALARY.
END JOBS RELATION.
!
!
! Departments Relation:
!
DEFINE RELATION DEPARTMENTS.
    DEPARTMENT_CODE.
    DEPARTMENT_NAME.
    MANAGER_ID
        BASED ON ID_NUMBER.
    BUDGET_PROJECTED
        BASED ON BUDGET.
    BUDGET_ACTUAL
        BASED ON BUDGET.
END DEPARTMENTS RELATION.
!
!
! Colleges Relation:
!
DEFINE RELATION COLLEGES.
    COLLEGE_CODE.
    COLLEGE_NAME.
    CITY.
    STATE.
    POSTAL_CODE.
END COLLEGES RELATION.
!
!
! Degrees Relation:
!
DEFINE RELATION DEGREES.
    EMPLOYEE_ID
        BASED ON ID_NUMBER.
    COLLEGE_CODE.
    YEAR_GIVEN.
    DEGREE.
    DEGREE_FIELD.
END DEGREES RELATION.
!
!
```

```
! Work_Status Relation:
!
DEFINE RELATION WORK_STATUS.
   STATUS_CODE.
   STATUS_NAME.
   STATUS_TYPE.
END WORK_STATUS RELATION.
!
!
! Resumes Relation:
!
DEFINE RELATION RESUMES.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   RESUME.
END RESUMES RELATION.
!
COMMIT
!
!
!************************************************************
!
!
! *** Define three views to get current information ***
!
! START_TRANSACTION READ_WRITE
!
!
! Current job information
!
DEFINE VIEW CURRENT_JOB OF JH IN JOB_HISTORY
  CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
     WITH JH.JOB_END MISSING.
      E.LAST_NAME.
      E.FIRST_NAME.
      E.EMPLOYEE_ID.
      JH.JOB_CODE.
      JH.DEPARTMENT_CODE.
      JH.SUPERVISOR_ID.
      JH.JOB_START.
END VIEW.
!
!
! Current salary information
!
DEFINE VIEW CURRENT_SALARY OF SH IN SALARY_HISTORY
 CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
   WITH SH.SALARY_END MISSING.
     E.LAST_NAME.
     E.FIRST_NAME.
     E.EMPLOYEE_ID.
     SH.SALARY_START.
     SH.SALARY_AMOUNT.
END VIEW.
!
!
```

(continued on next page)

```
! Current salary and job information
!
DEFINE VIEW CURRENT_INFO OF CJ IN CURRENT_JOB
  CROSS D IN DEPARTMENTS OVER DEPARTMENT_CODE
  CROSS J IN JOBS OVER JOB_CODE
  CROSS CS IN CURRENT_SALARY OVER EMPLOYEE_ID.
     LAST_NAME FROM CJ.LAST_NAME.
     FIRST_NAME FROM CJ.FIRST_NAME.
     ID FROM CJ.EMPLOYEE_ID.
     DEPARTMENT FROM D.DEPARTMENT_NAME.
     JOB FROM J.JOB_TITLE.
     JSTART FROM CJ.JOB_START.
     SSTART FROM CS.SALARY_START.
     SALARY FROM CS.SALARY_AMOUNT.
END VIEW.
!
COMMIT
!
!
! *** Define indexes for PERSONNEL ***
!
START_TRANSACTION READ_WRITE
!
!
! Index for EMPLOYEES:
!
DEFINE INDEX EMP_EMPLOYEE_ID FOR EMPLOYEES
   DUPLICATES ARE NOT ALLOWED.
      EMPLOYEE_ID.
END EMP_EMPLOYEE_ID INDEX.
!
!
! Index for JOB_HISTORY:
!
DEFINE INDEX JH_EMPLOYEE_ID FOR JOB_HISTORY
   DUPLICATES ARE ALLOWED.
      EMPLOYEE_ID.
END JH_EMPLOYEE_ID INDEX.
!
!
! Index for SALARY_HISTORY:
!
DEFINE INDEX SH_EMPLOYEE_ID FOR SALARY_HISTORY
   DUPLICATES ARE ALLOWED.
      EMPLOYEE_ID.
END SH_EMPLOYEE_ID INDEX.
!
!
! Indexes for DEGREES:
!
DEFINE INDEX DEG_COLLEGE_CODE FOR DEGREES
   DUPLICATES ARE ALLOWED.
      COLLEGE_CODE.
END DEG_COLLEGE_CODE INDEX.
!
DEFINE INDEX DEG_EMP_ID FOR DEGREES
   DUPLICATES ARE ALLOWED.
      EMPLOYEE_ID.
END DEG_EMP_ID INDEX.
```

```
!
!
! Index for COLLEGES:
!
DEFINE INDEX COLL_COLLEGE_CODE FOR COLLEGES
   DUPLICATES ARE NOT ALLOWED.
      COLLEGE_CODE.
END COLL_COLLEGE_CODE INDEX.
!
!
COMMIT
!
!
! *** Define constraints to validate field values
!
! START_TRANSACTION READ_WRITE
!
!
! Employee ID from JOB_HISTORY must exist in EMPLOYEES
!   relation before it can be stored in JOB_HISTORY
!
DEFINE CONSTRAINT JH_EMP_ID_EXISTS
   FOR JH IN JOB_HISTORY
   REQUIRE ANY E IN EMPLOYEES WITH
      E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   CHECK ON COMMIT.
!
!
! Employee ID from SALARY_HISTORY must exist in EMPLOYEES
!   relation before it can be stored in SALARY_HISTORY
!
DEFINE CONSTRAINT SH_EMP_ID_EXISTS
   FOR SH IN SALARY_HISTORY
   REQUIRE ANY E IN EMPLOYEES WITH
      E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   CHECK ON COMMIT.
!
!
! There must be an EMPLOYEE_ID (primary key) for each EMPLOYEE record
!
DEFINE CONSTRAINT EMPLOYEE_ID_REQUIRED
   FOR E IN EMPLOYEES
   REQUIRE NOT E.EMPLOYEE_ID MISSING.
!
!
! There must be a DEPARTMENT_CODE (primary key) for each DEPARTMENT record
!
DEFINE CONSTRAINT DEPT_CODE_REQUIRED
   FOR D IN DEPARTMENTS
   REQUIRE NOT D.DEPARTMENT_CODE MISSING.
!
!
! There must be JOB_CODE (primary key) for each JOBS record
!
DEFINE CONSTRAINT JOB_CODE_REQUIRED
   FOR J IN JOBS
   REQUIRE NOT J.JOB_CODE MISSING.
```

(continued on next page)

```
!
!
! There must be COLLEGE_CODE (primary key) for each COLLEGES record
!
DEFINE CONSTRAINT COLLEGE_CODE_REQUIRED
   FOR C IN COLLEGES
   REQUIRE NOT C.COLLEGE_CODE MISSING.
!
!-------------------------------------------
! Note that these constraints assume a certain order for loading
! data.  EMPLOYEES data must be loaded before JOB_HISTORY can be
! loaded, and so on.
!-------------------------------------------
COMMIT
!
!        Store three Work Status Codes in WORK_STATUS relation
!
START_TRANSACTION READ_WRITE RESERVING WORK_STATUS FOR SHARED WRITE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="0";
W.STATUS_NAME="INACTIVE";
W.STATUS_TYPE="RECORD EXPIRED";END_STORE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="1";
W.STATUS_NAME="ACTIVE";
W.STATUS_TYPE="FULL TIME";END_STORE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="2";
W.STATUS_NAME="ACTIVE";
W.STATUS_TYPE="PART TIME";END_STORE
COMMIT
!
FINISH
EXIT
```

A.1.2 PERS_WORKSPACE Definition

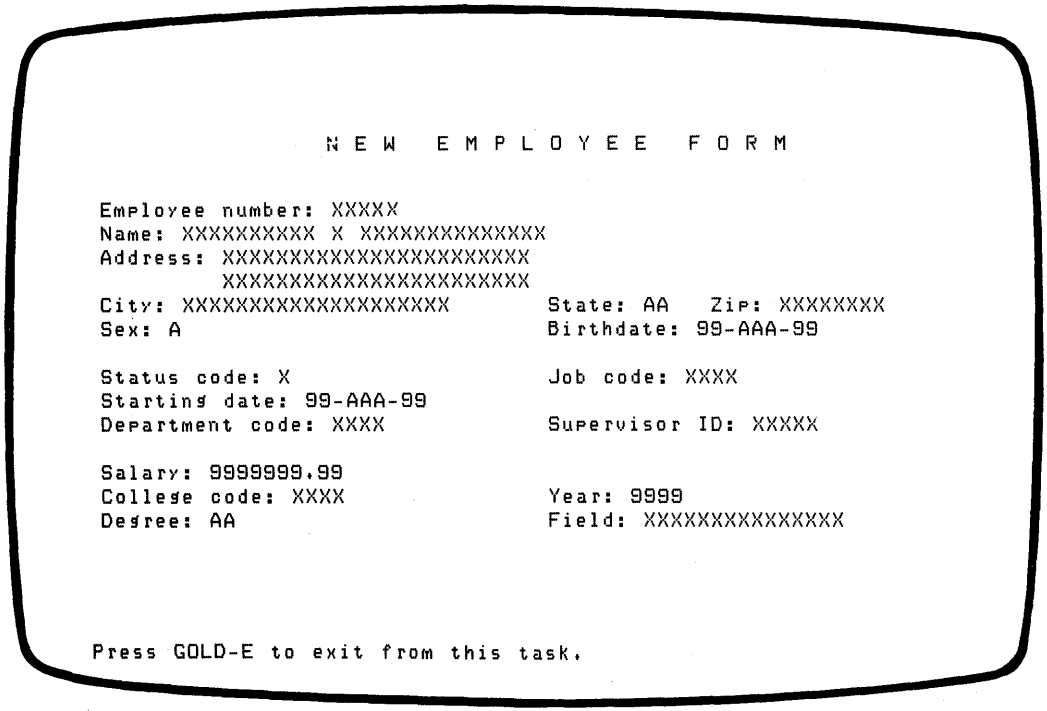
```
DEFINE RECORD PERS_WORKSPACE
   DESCRIPTION IS /* Miscellaneous fields */.

PERS_WORKSPACE STRUCTURE.
   PROGRAM_REQUEST_KEY              DATATYPE TEXT SIZE 6
                                    INITIAL_VALUE IS "      ".
   ERROR_FIELD                      DATATYPE TEXT SIZE 6
                                    INITIAL_VALUE IS "      ".
   NOT_FOUND                        DATATYPE TEXT SIZE 1
                                    INITIAL_VALUE IS " ".
   SAL_AMT                          DATATYPE SIGNED LONGWORD.
   JOB                              DATATYPE TEXT SIZE 4
                                    INITIAL_VALUE IS "    ".
   TEST_FIELD                       DATATYPE TEXT SIZE 1
                                    INITIAL_VALUE IS " ".
END PERS_WORKSPACE STRUCTURE.

END PERS_WORKSPACE.
```

A.1.3 Definitions for the Add Task

A.1.3.1 PERS_ADD_TASK Definition

```
REPLACE TASK PERS_ADD_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.DEGREES,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_ADD_REQUEST
         USING ACMS$PROCESSING_STATUS, DEGREES, EMPLOYEES, JOB_HISTORY,
         PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, DEGREES," &
      "JOB_HISTORY, SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_ADD IN PERS_SERVER
         USING DEGREES, EMPLOYEES, JOB_HISTORY, PERS_WORKSPACE,
         SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.3.2 PERS_ADD_FORM Definition



ZK-00051-00

A.1.3.3 PERS_ADD_REQUEST Definition

```
REPLACE REQUEST PERS_ADD_REQUEST

   FORM IS CDD$TOP.RDBPERS.PERS_ADD_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.DEGREES;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;
```

```
RECORD IS
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

DESCRIPTION /* Collect input for adding new employee to the database */;

USE FORM PERS_ADD_FORM;

INPUT EMP_NUMBER      TO EMPLOYEES.EMPLOYEE_ID,
      FIRST_NAME      TO FIRST_NAME,
      INITIAL         TO MIDDLE_INITIAL,
      LAST_NAME       TO LAST_NAME,
      ADDRESS1        TO ADDRESS_DATA_1,
      ADDRESS2        TO ADDRESS_DATA_2,
      CITY            TO CITY,
      STATE           TO STATE,
      POSTAL_CODE     TO POSTAL_CODE,
      SEX             TO SEX,
      BIRTHDAY        TO BIRTHDAY,
      STATUS_CODE     TO STATUS_CODE,
      JOB_CODE        TO JOB_CODE,
      JOB_START       TO JOB_START,
      DEPT_CODE       TO DEPARTMENT_CODE,
      SUPERVISOR_ID   TO SUPERVISOR_ID,
      SALARY          TO SALARY_AMOUNT,
      COLLEGE_CODE    TO COLLEGE_CODE,
      YEAR            TO YEAR_GIVEN,
      DEGREE          TO DEGREE,
      DEGREE_FIELD    TO DEGREE_FIELD;

PROGRAM KEY IS GOLD "E"
   NO CHECK;
   RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
END PROGRAM KEY;

CONTROL FIELD IS ACMS$T_STATUS_TYPE
   "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
END CONTROL FIELD;

END DEFINITION;
```

A.1.3.4 PERS_ADD Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_ADD.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB& INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"
```

(continued on next page)

```
01 DUP-EMP-NOS              PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_DUPEMPNOS.
01 REC-LOCKED               PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_RECLOCK.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_LOCK_CONFLICT       PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 RDB$_DEADLOCK            PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_NO_DUP              PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_NO_DUP.
01 LIB$SIGNAL               PIC S9(9) COMP
                              VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.DEGREES"
    FROM DICTIONARY
    REPLACING ==DEGREES. == BY ==DEGREES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
    FROM DICTIONARY
    REPLACING ==EMPLOYEES. == BY ==EMPLOYEES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING DEGREES_LINKAGE
                         EMPLOYEES_LINKAGE
                         JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program adds employee information to the Personnel database. Each
* employee has at least three records, one in each of the following
* relations: EMPLOYEES, JOB_HISTORY, and SALARY_HISTORY. If the employee
* has attended college, he or she also has a record in the DEGREES relatio

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Store the EMPLOYEES record
```

```
    &RDB& STORE E IN EMPLOYEES USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     E.EMPLOYEE_ID = EMPLOYEE_ID IN EMPLOYEES_LINKAGE;
    &RDB&     E.LAST_NAME = LAST_NAME IN EMPLOYEES_LINKAGE;
    &RDB&     E.FIRST_NAME = FIRST_NAME IN EMPLOYEES_LINKAGE;
    &RDB&     E.MIDDLE_INITIAL = MIDDLE_INITIAL IN EMPLOYEES_LINKAGE;
    &RDB&     E.ADDRESS_DATA_1 = ADDRESS_DATA_1 IN EMPLOYEES_LINKAGE;
    &RDB&     E.ADDRESS_DATA_2 = ADDRESS_DATA_2 IN EMPLOYEES_LINKAGE;
    &RDB&     E.CITY = CITY IN EMPLOYEES_LINKAGE;
    &RDB&     E.STATE = STATE IN EMPLOYEES_LINKAGE;
    &RDB&     E.POSTAL_CODE = POSTAL_CODE IN EMPLOYEES_LINKAGE;
    &RDB&     E.SEX = SEX IN EMPLOYEES_LINKAGE;
    &RDB&     E.BIRTHDAY = BIRTHDAY IN EMPLOYEES_LINKAGE;
    &RDB&     E.STATUS_CODE = STATUS_CODE IN EMPLOYEES_LINKAGE
    &RDB& END_STORE

* Store the JOB_HISTORY record

    &RDB& STORE JH IN JOB_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB& END_ERROR
    &RDB&   JH.EMPLOYEE_ID = EMPLOYEE_ID IN EMPLOYEES_LINKAGE;
    &RDB&   JH.JOB_CODE = JOB_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&   JH.JOB_START = JOB_START IN JOB_HISTORY_LINKAGE;
    &RDB&   JH.DEPARTMENT_CODE = DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&   JH.SUPERVISOR_ID = SUPERVISOR_ID IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE

* Store the SALARY_HISTORY record

    &RDB& STORE SH IN SALARY_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     SH.EMPLOYEE_ID = EMPLOYEE_ID IN EMPLOYEES_LINKAGE;
    &RDB&     SH.SALARY_AMOUNT = SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE;
    &RDB&     SH.SALARY_START = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE

* If the employee attended college, store the DEGREES record.

    IF DEGREE OF DEGREES_LINKAGE NOT = SPACES
    THEN
    &RDB& STORE D IN DEGREES USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
```

(continued on next page)

```
    &RDB&      D.EMPLOYEE_ID = EMPLOYEE_ID IN EMPLOYEES_LINKAGE;
    &RDB&      D.COLLEGE_CODE = COLLEGE_CODE;
    &RDB&      D.YEAR_GIVEN = YEAR_GIVEN;
    &RDB&      D.DEGREE = DEGREE;
    &RDB&      D.DEGREE_FIELD = DEGREE_FIELD
    &RDB& END_STORE
    END-IF.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record and duplicate employee number are the
* expected errors. Signal any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       IF RDB$STATUS EQUAL RDB$_NO_DUP
       THEN
          MOVE DUP-EMP-NOS TO STATUS-RESULT
       ELSE
          MOVE DB-FAILURE TO STATUS-RESULT
          CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                                 BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.4 Definitions for the Display Task

A.1.4.1 PERS_DISPLAY_TASK Definition

```
REPLACE TASK PERS_DISPLAY_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.CURRENT_INFO,
   CDD$TOP.RDBPERS.PERS_WORKSPACE;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_DISPLAY_REQUEST1
         USING ACMS$PROCESSING_STATUS, CURRENT_INFO, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
       END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_DISPLAY IN PERS_SERVER
         USING CURRENT_INFO, PERS_WORKSPACE;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS PERS_DISPLAY_REQUEST2
         USING CURRENT_INFO, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "REPEAT" : REPEAT TASK;
         "EXIT"   : EXIT TASK;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.4.2 PERS_DISPLAY_FORM Definition

```
                D I S P L A Y   E M P L O Y E E   F O R M

    Employee number: XXXXX

    Name: XXXXXXXXXX X XXXXXXXXXXXXXX
    Department: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    Job title: XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    Job starting date: 99-AAA-99

    Salary starting date: 99-AAA-99
    Salary: 9999999.99




  Press GOLD-E to exit from this task.
```

ZK-00052-00

A.1.4.3 PERS_DISPLAY_REQUEST1 Definition

```
REPLACE REQUEST PERS_DISPLAY_REQUEST1

   FORM IS CDD$TOP.RDBPERS.PERS_DISPLAY_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.CURRENT_INFO;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Collect the employee ID number for retrieving
                  employee data for display */;

   USE FORM PERS_DISPLAY_FORM;
```

```
   INPUT EMP_NUMBER TO ID;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

END DEFINITION;
```

A.1.4.4 PERS_DISPLAY_REQUEST2 Definition

```
REPLACE REQUEST PERS_DISPLAY_REQUEST2

   FORM IS CDD$TOP.RDBPERS.PERS_DISPLAY_FORM;

   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.CURRENT_INFO;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Display basic employee data */;

   USE FORM PERS_DISPLAY_FORM;

   OUTPUT ID                TO EMP_NUMBER,
          FIRST_NAME        TO FIRST_NAME,
          LAST_NAME         TO LAST_NAME,
          DEPARTMENT        TO DEPT_NAME,
          CURRENT_INFO.JOB  TO JOB_TITLE,
          JSTART            TO JOB_START,
          SSTART            TO SALARY_START,
          SALARY            TO SALARY;

   WAIT;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   PROGRAM KEY IS GOLD "R"
      NO CHECK;
      RETURN "REPEAT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   END DEFINITION;
```

A.1.4.5 PERS_GET_DISPLAY Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_GET_DISPLAY.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED               PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK            PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT       PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL               PIC S9(9) COMP
                              VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.CURRENT_INFO"
    FROM DICTIONARY
    REPLACING ==CURRENT_INFO. == BY ==CURRENT_INFO_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION USING CURRENT_INFO_LINKAGE
                         PERS_WORKSPACE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program gets information through the CURRENT_INFO view for display.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Get a record from the CURRENT_INFO relation based on the employee ID.

    &RDB& FOR C IN CURRENT_INFO WITH C.ID = ID IN CURRENT_INFO_LINKAGE
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
```

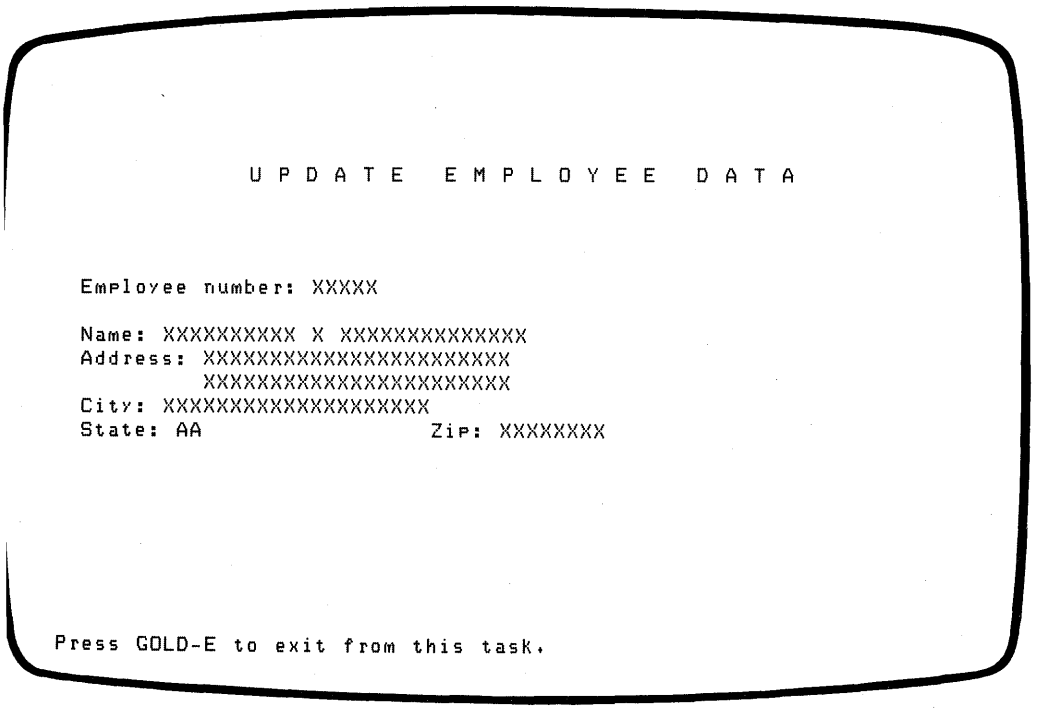
```
    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       LAST_NAME IN CURRENT_INFO_LINKAGE = C.LAST_NAME;
    &RDB&       FIRST_NAME IN CURRENT_INFO_LINKAGE = C.FIRST_NAME;
    &RDB&       DEPARTMENT IN CURRENT_INFO_LINKAGE = C.DEPARTMENT;
    &RDB&       JOB IN CURRENT_INFO_LINKAGE = C.JOB;
    &RDB&       JSTART IN CURRENT_INFO_LINKAGE = C.JSTART;
    &RDB&       SSTART IN CURRENT_INFO_LINKAGE = C.SSTART;
    &RDB&       SALARY IN CURRENT_INFO_LINKAGE = C.SALARY
    &RDB&   END_GET
    &RDB& END_FOR

* If the employee ID is not in the CURRENT_INFO relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
      MOVE DB-FAILURE TO STATUS-RESULT
      CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                             BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.5 Definitions for the General Update Task

A.1.5.1 PERS_UPDATE_GENERAL_TASK Definition

```
REPLACE TASK PERS_UPDATE_GENERAL_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES,
   CDD$TOP.RDBPERS.PERS_WORKSPACE;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_UPDATE_GENERAL_REQUEST1
         USING ACMS$PROCESSING_STATUS, EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_EMPLOYEE IN PERS_SERVER
         USING EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS PERS_UPDATE_GENERAL_REQUEST2
         USING ACMS$PROCESSING_STATUS, EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
      "SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_UPDATE_EMPLOYEE IN PERS_SERVER
         USING EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.5.2 PERS_UPDATE_GENERAL_FORM Definition



ZK-00053-00

A.1.5.3 PERS_UPDATE_GENERAL_REQUEST1 Definition

```
REPLACE REQUEST PERS_UPDATE_GENERAL_REQUEST1

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_GENERAL_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Collect the employee ID number for retrieving
                  employee data for update */;

   USE FORM PERS_UPDATE_GENERAL_FORM;
```

(continued on next page)

```
   INPUT EMP_NUMBER TO EMPLOYEE_ID;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.1.5.4 PERS_UPDATE_GENERAL_REQUEST2 Definition

```
REPLACE REQUEST PERS_UPDATE_GENERAL_REQUEST2;

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_GENERAL_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Display general employee data and accept changes */;

   USE FORM PERS_UPDATE_GENERAL_FORM;

   OUTPUT LAST_NAME       TO LAST_NAME,
          FIRST_NAME      TO FIRST_NAME,
          MIDDLE_INITIAL  TO INITIAL,
          ADDRESS_DATA_1  TO ADDRESS1,
          ADDRESS_DATA_2  TO ADDRESS2,
          CITY            TO CITY,
          STATE           TO STATE,
          POSTAL_CODE     TO POSTAL_CODE;

   INPUT LAST_NAME     TO LAST_NAME,
         FIRST_NAME    TO FIRST_NAME,
         INITIAL       TO MIDDLE_INITIAL,
         ADDRESS1      TO ADDRESS_DATA_1,
         ADDRESS2      TO ADDRESS_DATA_2,
         CITY          TO CITY,
         STATE         TO STATE,
         POSTAL_CODE   TO POSTAL_CODE;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

END DEFINITION;
```

A.1.5.5 PERS_GET_EMPLOYEE Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_GET_EMPLOYEE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT      PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL              PIC S9(9) COMP
                             VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT           PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
    FROM DICTIONARY
    REPLACING ==EMPLOYEES. == BY ==EMPLOYEES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION USING EMPLOYEES_LINKAGE
                         PERS_WORKSPACE
                   GIVING STATUS-RESULT.

MAIN SECTION.
OOO-MAIN-PARAGRAPH.

* This program retrieves an EMPLOYEES record that is then displayed on
* a form, where the user can modify the record's contents.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

(continued on next page)

```
* Get basic data on an employee.

    &RDB& FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&    GET
    &RDB&      ON ERROR
                 PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                 GO TO 100-EXIT-PROGRAM
    &RDB&      END_ERROR
    &RDB&        LAST_NAME IN EMPLOYEES_LINKAGE = E.LAST_NAME;
    &RDB&        FIRST_NAME IN EMPLOYEES_LINKAGE = E.FIRST_NAME;
    &RDB&        MIDDLE_INITIAL IN EMPLOYEES_LINKAGE = E.MIDDLE_INITIAL;
    &RDB&        ADDRESS_DATA_1 IN EMPLOYEES_LINKAGE = E.ADDRESS_DATA_1;
    &RDB&        ADDRESS_DATA_2 IN EMPLOYEES_LINKAGE = E.ADDRESS_DATA_2;
    &RDB&        CITY IN EMPLOYEES_LINKAGE = E.CITY;
    &RDB&        STATE IN EMPLOYEES_LINKAGE = E.STATE;
    &RDB&        POSTAL_CODE IN EMPLOYEES_LINKAGE = E.POSTAL_CODE
    &RDB&   END_GET
    &RDB& END_FOR

* If the employee ID was not found in the database, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.5.6 PERS_UPDATE_EMPLOYEE Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_UPDATE_EMPLOYEE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED                PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND             PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE                PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK             PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT        PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL                PIC S9(9) COMP
                               VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT             PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
    FROM DICTIONARY
    REPLACING ==EMPLOYEES. == BY ==EMPLOYEES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION USING EMPLOYEES_LINKAGE
                         PERS_WORKSPACE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program writes a modified EMPLOYEES record to the database.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Modify the EMPLOYEES record

    &RDB& FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
```

(continued on next page)

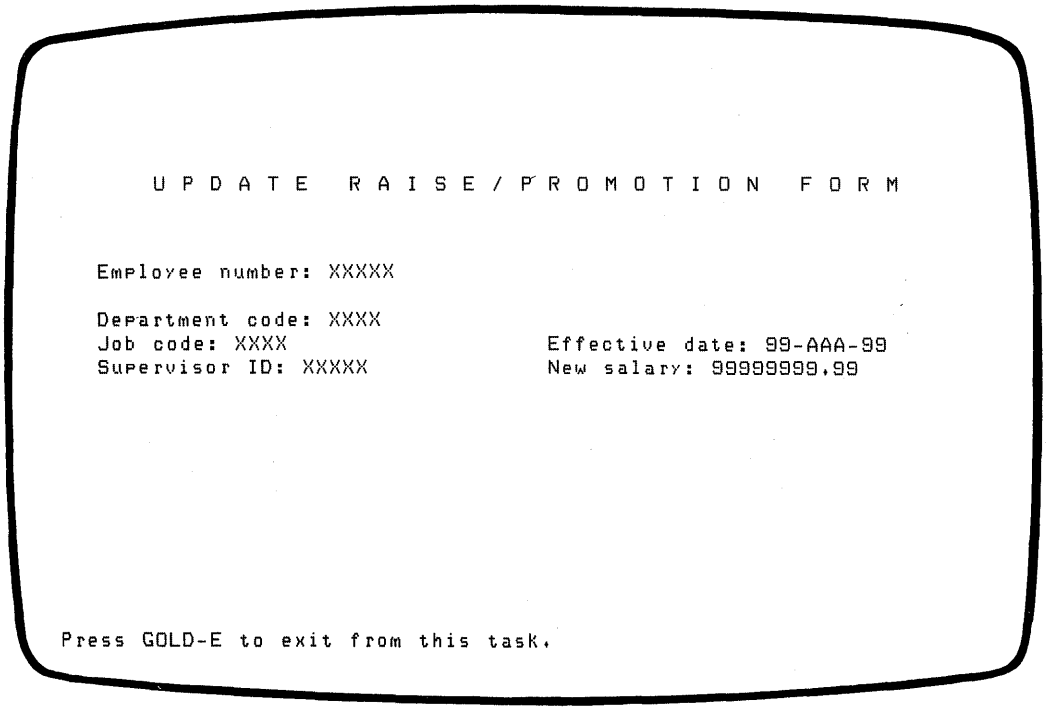
```
    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY E USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       E.LAST_NAME = LAST_NAME IN EMPLOYEES_LINKAGE;
    &RDB&       E.FIRST_NAME = FIRST_NAME IN EMPLOYEES_LINKAGE;
    &RDB&       E.MIDDLE_INITIAL = MIDDLE_INITIAL IN EMPLOYEES_LINKAGE;
    &RDB&       E.ADDRESS_DATA_1 = ADDRESS_DATA_1 IN EMPLOYEES_LINKAGE;
    &RDB&       E.ADDRESS_DATA_2 = ADDRESS_DATA_2 IN EMPLOYEES_LINKAGE;
    &RDB&       E.CITY = CITY IN EMPLOYEES_LINKAGE;
    &RDB&       E.STATE = STATE IN EMPLOYEES_LINKAGE;
    &RDB&       E.POSTAL_CODE = POSTAL_CODE IN EMPLOYEES_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR

* If the employee ID is not in the EMPLOYEES relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.6 Definitions for the Raise/Promotion Update Task

A.1.6.1 PERS_UPDATE_RAISEPRO_TASK Definition

```
REPLACE TASK PERS_UPDATE_RAISEPRO_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST1
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_RAISEPRO IN PERS_SERVER
         USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST2
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE,
         SALARY_HISTORY;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
      "SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_UPDATE_RAISEPRO IN PERS_SERVER
          USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
          "B" : GET ERROR MESSAGE;
                ROLLBACK;
                GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.6.2 PERS_UPDATE_RAISEPRO_FORM Definition



ZK-00046-00

A.1.6.3 PERS_UPDATE_RAISEPRO_REQUEST1 Definition

```
REPLACE REQUEST PERS_UPDATE_RAISEPRO_REQUEST1

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Accept the employee ID number for retrieving
                  job and salary information for raise/promotion
                  update */;

   USE FORM PERS_UPDATE_RAISEPRO_FORM;
```

```
   INPUT EMP_NUMBER TO EMPLOYEE_ID;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.1.6.4 PERS_UPDATE_RAISEPRO_REQUEST2 Definition

```
REPLACE REQUEST PERS_UPDATE_RAISEPRO_REQUEST2

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_RAISEPRO_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

   DESCRIPTION /* Display job and salary information and accept
                  changes to indicate a raise and/or a promotion */;

   USE FORM PERS_UPDATE_RAISEPRO_FORM;

   OUTPUT DEPARTMENT_CODE          TO DEPT_CODE,
          JOB_CODE                 TO JOB_CODE,
          SUPERVISOR_ID            TO SUPERVISOR_ID,
          SALARY_AMOUNT            TO SALARY;

   INPUT JOB_CODE      TO JOB_CODE,
         SUPERVISOR_ID TO SUPERVISOR_ID,
         SALARY        TO SALARY_AMOUNT;

   RETURN JOB_START TO JOB_START;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.1.6.5 PERS_GET_RAISEPRO Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_GET_RAISEPRO.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT      PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL              PIC S9(9) COMP
                             VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT           PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program retrieves JOB_HISTORY and SALARY_HISTORY records that are
* then displayed on a form where the user can record a raise or promotion.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

```
* Get job history information for an employee

    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   JH.JOB_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JOB_CODE IN JOB_HISTORY_LINKAGE = JH.JOB_CODE;
    &RDB&       DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE = JH.DEPARTMENT_CODE;
    &RDB&       JOB_START IN JOB_HISTORY_LINKAGE = JH.JOB_START;
    &RDB&       SUPERVISOR_ID IN JOB_HISTORY_LINKAGE = JH.SUPERVISOR_ID
    &RDB&   END_GET
    &RDB& END_FOR

* If employee ID is not in the JOB_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Reset the record-found flag

    MOVE "T" TO NOT_FOUND.

* Save the old job code for comparison in the update procedure.

    MOVE JOB_CODE OF JOB_HISTORY_LINKAGE TO JOB
        OF PERS_WORKSPACE.

* Get salary history information for an employee

    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   SH.SALARY_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND
```

(continued on next page)

```
    &RDB&    GET
    &RDB&      ON ERROR
                 PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                 GO TO 100-EXIT-PROGRAM
    &RDB&      END_ERROR
    &RDB&        SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE = SH.SALARY_AMOUNT
    &RDB&    END_GET
    &RDB& END_FOR

* Save the old salary for comparison in the update procedure

    MOVE SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE TO
        SAL_AMT OF PERS_WORKSPACE.

* If the employee ID is not in the SALARY_HISTORY relation, return an error

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.6.6 PERS_UPDATE_RAISEPRO Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_UPDATE_RAISEPRO.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT      PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL              PIC S9(9) COMP
                             VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT           PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                  GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program writes modified JOB_HISTORY and SALARY_HISTORY records to
* the database.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

(continued on next page)

```
* If the user changed the job code, fill in the job ending date in the
* JOB_HISTORY relation and store a new JOB_HISTORY record for the new job.

    IF JOB_CODE OF JOB_HISTORY_LINKAGE = JOB OF PERS_WORKSPACE
    THEN
        MOVE "F" TO NOT_FOUND
    ELSE
    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   JH.JOB_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY JH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JH.JOB_END = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR

    &RDB& STORE JH IN JOB_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     JH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.JOB_CODE = JOB_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.DEPARTMENT_CODE = DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.JOB_START = JOB_START IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.SUPERVISOR_ID = SUPERVISOR_ID IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE
    END-IF.

* If the employee ID is not in the JOB_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Reset the record-found flag

    MOVE "T" TO NOT_FOUND.

* If the user changed the salary, fill in the salary ending date in the
* SALARY_HISTORY relation and store a new SALARY_HISTORY record for the
* new salary.

    IF SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE =
        SAL_AMT OF PERS_WORKSPACE
    THEN
        MOVE "F" TO NOT_FOUND
```

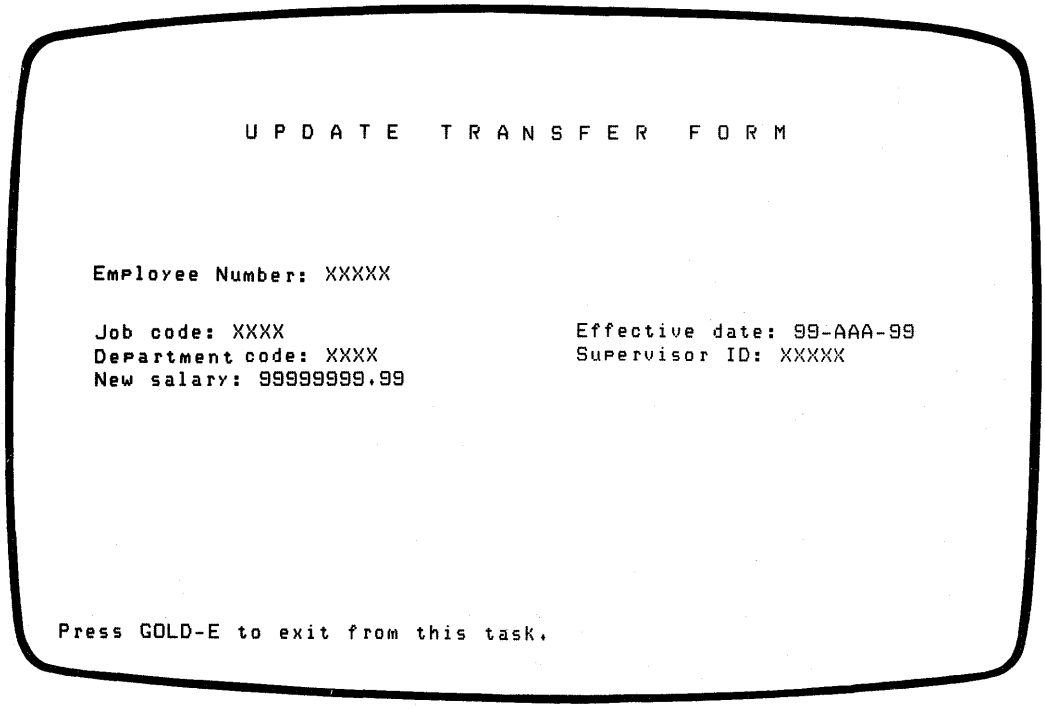
```
    ELSE
    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   SH.SALARY_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY SH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       SH.SALARY_END = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR

    &RDB& STORE SH IN SALARY_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     SH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
    &RDB&     SH.SALARY_AMOUNT = SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE;
    &RDB&     SH.SALARY_START = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE
    END-IF.

* If the employee ID is not in the SALARY_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.7 Definitions for the Transfer Update Task

A.1.7.1 PERS_UPDATE_TRANSFER_TASK Definition

```
REPLACE TASK PERS_UPDATE_TRANSFER_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_UPDATE_TRANSFER_REQUEST1
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_TRANSFER IN PERS_SERVER
         USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS PERS_UPDATE_TRANSFER_REQUEST2
         USING ACMS$PROCESSING_STATUS, JOB_HISTORY, PERS_WORKSPACE,
         SALARY_HISTORY;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
      "SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_UPDATE_TRANSFER IN PERS_SERVER
         USING JOB_HISTORY, PERS_WORKSPACE, SALARY_HISTORY;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.7.2 PERS_UPDATE_TRANSFER_FORM Definition



ZK-00054-00

A.1.7.3 PERS_UPDATE_TRANSFER_REQUEST1 Definition

```
REPLACE REQUEST PERS_UPDATE_TRANSFER_REQUEST1

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_TRANSFER_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Accept the employee ID number for retrieving
                  job and salary information for transfer update */;

   USE FORM PERS_UPDATE_TRANSFER_FORM;
```

(continued on next page)

```
   INPUT EMP_NUMBER TO EMPLOYEE_ID;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.1.7.4 PERS_UPDATE_TRANSFER_REQUEST2 Definition

```
REPLACE REQUEST PERS_UPDATE_TRANSFER_REQUEST2

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_TRANSFER_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY;

   DESCRIPTION /* Display job and salary information and accept
                  changes to indicate a transfer */;

   USE FORM PERS_UPDATE_TRANSFER_FORM;

   OUTPUT DEPARTMENT_CODE          TO DEPT_CODE,
          SUPERVISOR_ID            TO SUPERVISOR_ID,
          JOB_CODE                 TO JOB_CODE,
          SALARY_AMOUNT            TO SALARY;

   INPUT DEPT_CODE     TO DEPARTMENT_CODE,
         SUPERVISOR_ID TO SUPERVISOR_ID,
         JOB_CODE      TO JOB_CODE,
         SALARY        TO SALARY_AMOUNT;

   RETURN JOB_START TO JOB_START;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.1.7.5 PERS_GET_TRANSFER Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.     PERS_GET_TRANSFER.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT      PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL              PIC S9(9) COMP
                             VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT           PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program retrieves JOB_HISTORY and SALARY_HISTORY records that are
* then displayed on a form where the user can record a transfer.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

(continued on next page)

```
* Get job history information for an employee

    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   JH.JOB_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JOB_CODE IN JOB_HISTORY_LINKAGE = JH.JOB_CODE;
    &RDB&       DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE = JH.DEPARTMENT_COD]
    &RDB&       JOB_START IN JOB_HISTORY_LINKAGE = JH.JOB_START;
    &RDB&       SUPERVISOR_ID IN JOB_HISTORY_LINKAGE = JH.SUPERVISOR_ID
    &RDB&   END_GET
    &RDB& END_FOR

* If employee ID is not in the JOB_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Reset record-found flag

    MOVE "T" TO NOT_FOUND.

* Get salary history information for an employee

    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   SH.SALARY_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE = SH.SALARY_AMOUNT
    &RDB&   END_GET
    &RDB& END_FOR
```

```
* If employee ID is not in the SALARY_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Save the old salary for comparison in the update procedure.

    MOVE SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE TO
        SAL_AMT OF PERS_WORKSPACE.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.7.6 PERS_UPDATE_TRANSFER Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_UPDATE_TRANSFER.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&  INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND           PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE              PIC S9(9) COMP
                             VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK           PIC S9(9) COMP
                             VALUE IS EXTERNAL RDB$_DEADLOCK.
```

(continued on next page)

```
01 RDB$_LOCK_CONFLICT       PIC S9(9) COMP
                              VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL               PIC S9(9) COMP
                              VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.SALARY_HISTORY"
    FROM DICTIONARY
    REPLACING ==SALARY_HISTORY. == BY ==SALARY_HISTORY_LINKAGE. ==.

PROCEDURE DIVISION USING JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                         SALARY_HISTORY_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program writes modified JOB_HISTORY and SALARY_HISTORY records to
* the database.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Fill in the job ending date in the JOB_HISTORY relation and store a new
* JOB_HISTORY record for the new job.

    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   JH.JOB_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY JH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JH.JOB_END = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR
```

```
    &RDB& STORE JH IN JOB_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     JH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.JOB_CODE = JOB_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.JOB_START = JOB_START IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.DEPARTMENT_CODE = DEPARTMENT_CODE IN JOB_HISTORY_LINKAGE;
    &RDB&     JH.SUPERVISOR_ID = SUPERVISOR_ID IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE

* If the employee ID is not in the JOB_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Reset the record-found flag

    MOVE "T" TO NOT_FOUND.

* If the user changed the salary, fill in the salary ending date in the
* SALARY_HISTORY relation and store a new SALARY_HISTORY record for the
* new salary.

    IF SALARY_AMOUNT OF SALARY_HISTORY_LINKAGE =
        SAL_AMT OF PERS_WORKSPACE
    THEN
        MOVE "F" TO NOT_FOUND
    ELSE
    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN JOB_HISTORY_LINKAGE AND
    &RDB&   SH.SALARY_END MISSING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY SH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       SH.SALARY_END = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR
```

(continued on next page)

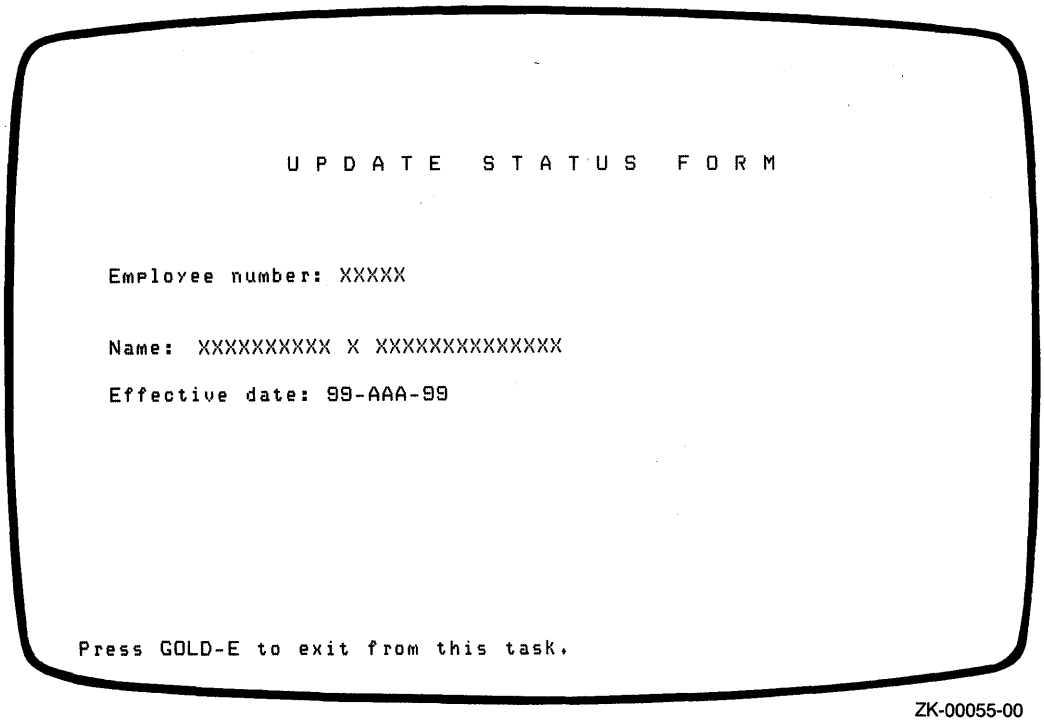
```
    &RDB& STORE SH IN SALARY_HISTORY USING
    &RDB&   ON ERROR
              PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
              GO TO 100-EXIT-PROGRAM
    &RDB&   END_ERROR
    &RDB&     SH.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY_LINKAGE;
    &RDB&     SH.SALARY_AMOUNT = SALARY_AMOUNT IN SALARY_HISTORY_LINKAGE;
    &RDB&     SH.SALARY_START = JOB_START IN JOB_HISTORY_LINKAGE
    &RDB& END_STORE
    END-IF.

* If the employee ID is not in the SALARY_HISTORY relation, return an error

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.8 Definitions for the Status Update Task

A.1.8.1 PERS_UPDATE_STATUS_TASK Definition

```
REPLACE TASK PERS_UPDATE_STATUS_TASK

WORKSPACES ARE
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES,
   CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY,
   CDD$TOP.RDBPERS.PERS_WORKSPACE;

BLOCK WORK
   EXCHANGE
      REQUEST IS PERS_UPDATE_STATUS_REQUEST1
         USING ACMS$PROCESSING_STATUS, EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY "START_TRANSACTION READ_ONLY"
      CALL PERS_GET_STATUS IN PERS_SERVER
         USING EMPLOYEES, PERS_WORKSPACE;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS PERS_UPDATE_STATUS_REQUEST2
         USING ACMS$PROCESSING_STATUS, EMPLOYEES, JOB_HISTORY,
         PERS_WORKSPACE;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH RDB RECOVERY
      "START_TRANSACTION READ_WRITE RESERVING EMPLOYEES, JOB_HISTORY," &
      "SALARY_HISTORY FOR SHARED WRITE"
      CALL PERS_UPDATE_STATUS IN PERS_SERVER
         USING EMPLOYEES, JOB_HISTORY, PERS_WORKSPACE;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.1.8.2 PERS_UPDATE_STATUS_FORM Definition



ZK-00055-00

A.1.8.3 PERS_UPDATE_STATUS_REQUEST1 Definition

```
REPLACE REQUEST PERS_UPDATE_STATUS_REQUEST1

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_STATUS_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Accept the employee ID number for retrieving
                  employee information for status update */;

   USE FORM PERS_UPDATE_STATUS_FORM;
```

```
   INPUT EMP_NUMBER TO EMPLOYEE_ID;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

END DEFINITION;
```

A.1.8.4 PERS_UPDATE_STATUS_REQUEST2 Definition

```
REPLACE REQUEST PERS_UPDATE_STATUS_REQUEST2

   FORM IS CDD$TOP.RDBPERS.PERS_UPDATE_STATUS_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES;
   RECORD IS
      CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY;
   RECORD IS
      CDD$TOP.RDBPERS.PERS_WORKSPACE;

   DESCRIPTION /* Display employee information and let user confirm
                  employee whose status is to be changed to inactive */;

   USE FORM PERS_UPDATE_STATUS_FORM;

   OUTPUT FIRST_NAME             TO FIRST_NAME,
          MIDDLE_INITIAL         TO INITIAL,
          LAST_NAME              TO LAST_NAME;

   OUTPUT 'Press RETURN to confirm.' to CONFIRM_FIELD;

   RETURN EFF_DATE TO JOB_END;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

END DEFINITION;
```

A.1.8.5 PERS_GET_STATUS Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_GET_STATUS.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&    INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-NOT-FOUND             PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_RECNOTFD.
01 REC-LOCKED                PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_RECLOCK.
01 DB-FAILURE                PIC S9(9) COMP
                               VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK             PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT        PIC S9(9) COMP
                               VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL                PIC S9(9) COMP
                               VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT             PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
    FROM DICTIONARY
    REPLACING ==EMPLOYEES. == BY ==EMPLOYEES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION USING EMPLOYEES_LINKAGE
                         PERS_WORKSPACE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program retrieves the employee name for display so that the user
* can verify the employee before changing the work status to inactive.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

```
* Get employee information

    &RDB& FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                  PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                  GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   GET
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       LAST_NAME IN EMPLOYEES_LINKAGE = E.LAST_NAME;
    &RDB&       FIRST_NAME IN EMPLOYEES_LINKAGE = E.FIRST_NAME;
    &RDB&       MIDDLE_INITIAL IN EMPLOYEES_LINKAGE = E.MIDDLE_INITIAL
    &RDB&   END_GET
    &RDB& END_FOR

* If the employee ID is not in the EMPLOYEES relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.8.6 PERS_UPDATE_STATUS Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    PERS_UPDATE_STATUS.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB&    INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01 REC-LOCKED                 PIC S9(9) COMP
                                VALUE IS EXTERNAL PRS_RECLOCK.
01 REC-NOT-FOUND              PIC S9(9) COMP
                                VALUE IS EXTERNAL PRS_RECNOTFD.
01 DB-FAILURE                 PIC S9(9) COMP
                                VALUE IS EXTERNAL PRS_DBFAIL.
01 RDB$_DEADLOCK              PIC S9(9) COMP
                                VALUE IS EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT         PIC S9(9) COMP
                                VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
01 LIB$SIGNAL                 PIC S9(9) COMP
                                VALUE IS EXTERNAL LIB$SIGNAL.
01 STATUS-RESULT              PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
    FROM DICTIONARY
    REPLACING ==EMPLOYEES. == BY ==EMPLOYEES_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERSONNEL.RDB$RELATIONS.JOB_HISTORY"
    FROM DICTIONARY
    REPLACING ==JOB_HISTORY. == BY ==JOB_HISTORY_LINKAGE. ==.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION USING EMPLOYEES_LINKAGE
                         JOB_HISTORY_LINKAGE
                         PERS_WORKSPACE
                   GIVING STATUS-RESULT.

MAIN SECTION.
000-MAIN-PARAGRAPH.

* This program writes modified EMPLOYEES, JOB_HISTORY, and SALARY_HISTORY
* records to the database.

    SET STATUS-RESULT TO SUCCESS.

    MOVE "T" TO NOT_FOUND.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

```
* Change the STATUS_CODE field in the EMPLOYEES relation to 0 to
* indicate an inactive status.

    &RDB& FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY E USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       E.STATUS_CODE = 0
    &RDB&   END_MODIFY
    &RDB& END_FOR

* If the employee ID is not in the EMPLOYEES relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.

* Reset record-found flag

    MOVE "T" TO NOT_FOUND.

* Fill in job ending date in JOB_HISTORY relation

    &RDB& FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY JH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       JH.JOB_END = JOB_END IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR

* If employee ID is not in the JOB_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT
        GO TO 100-EXIT-PROGRAM.
```

(continued on next page)

```
* Reset record-found flag

    MOVE "T" TO NOT_FOUND.

* Fill in salary ending date in SALARY_HISTORY relation

    &RDB& FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID =
    &RDB&   EMPLOYEE_ID IN EMPLOYEES_LINKAGE
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR

    MOVE "F" TO NOT_FOUND

    &RDB&   MODIFY SH USING
    &RDB&     ON ERROR
                PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT
                GO TO 100-EXIT-PROGRAM
    &RDB&     END_ERROR
    &RDB&       SH.SALARY_END = JOB_END IN JOB_HISTORY_LINKAGE
    &RDB&   END_MODIFY
    &RDB& END_FOR

* If employee ID is not in the SALARY_HISTORY relation, return an error.

    IF NOT_FOUND = "T"
    THEN
        MOVE REC-NOT-FOUND TO STATUS-RESULT.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* Test for errors. Locked record is the only expected error.  Signal
* any other errors.

    IF RDB$STATUS EQUAL RDB$_DEADLOCK
       OR RDB$STATUS EQUAL RDB$_LOCK_CONFLICT
    THEN
       MOVE REC-LOCKED TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                              BY VALUE LIB$SIGNAL.

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.9 Server Procedures

A.1.9.1 Initialization Procedure

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    PERS_STARTUP.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB& INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01  DB-FAILURE             PIC S9(9) COMP
                              VALUE IS EXTERNAL PRS_DBFAIL.
01  LIB$SIGNAL             PIC S9(9) COMP
                              VALUE IS EXTERNAL LIB$SIGNAL.
01  STATUS-RESULT          PIC S9(9) COMP.

COPY "CDD$TOP.RDBPERS.PERS_WORKSPACE" FROM DICTIONARY.

PROCEDURE DIVISION GIVING STATUS-RESULT.

MAIN SECTION.
000-START.

* Start transaction and read a WORK_STATUS record.  The overhead
* associated with the first database access is thus incurred in
* the initialization procedure and not in the first selected task.

    SET STATUS-RESULT TO SUCCESS.

    &RDB& START_TRANSACTION READ_WRITE
    &RDB&   ON ERROR
              CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                                     BY VALUE LIB$SIGNAL
    &RDB&   END_ERROR

    &RDB& FOR WS IN WORK_STATUS
    &RDB&   GET
    &RDB&     ON ERROR
                MOVE DB-FAILURE TO STATUS-RESULT
                CALL "LIB$CALLG" USING BY REFERENCE RDB$MESSAGE_VECTOR
                                       BY VALUE LIB$SIGNAL
    &RDB&     END_ERROR
    &RDB&       TEST_FIELD IN PERS_WORKSPACE = WS.STATUS_CODE
    &RDB&   END_GET
    &RDB& END_FOR

    &RDB& COMMIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.9.2 Termination Procedure

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    PERS_SHUTDOWN.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.

WORKING-STORAGE SECTION.

    &RDB& INVOKE DATABASE FILENAME "PERS$EXE:PERSONNEL"

01  STATUS-RESULT          PIC S9(9) COMP.

PROCEDURE DIVISION GIVING STATUS-RESULT.
MAIN SECTION.
000-START.

* Finish the database when the server is stopped.

    SET STATUS-RESULT TO SUCCESS.

    &RDB& FINISH.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.1.10 Request Library Definition

```
REPLACE LIBRARY PERS_REQLIB

REQUEST IS PERS_ADD_REQUEST;
REQUEST IS PERS_DISPLAY_REQUEST1;
REQUEST IS PERS_DISPLAY_REQUEST2;
REQUEST IS PERS_UPDATE_GENERAL_REQUEST1;
REQUEST IS PERS_UPDATE_GENERAL_REQUEST2;
REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST1;
REQUEST IS PERS_UPDATE_RAISEPRO_REQUEST2;
REQUEST IS PERS_UPDATE_TRANSFER_REQUEST1;
REQUEST IS PERS_UPDATE_TRANSFER_REQUEST2;
REQUEST IS PERS_UPDATE_STATUS_REQUEST1;
REQUEST IS PERS_UPDATE_STATUS_REQUEST2;

END DEFINITION;
```

A.1.11 Task Group Definition

```
REPLACE GROUP PERS_TASK_GROUP

   REQUEST LIBRARY IS "PERS$EXE:PERS_REQLIB.RLB";

   MESSAGE FILE IS "PERS$EXE:PERSMSG.EXE";

   DEFAULT TASK GROUP FILE IS "PERS$EXE:PERS_GROUP.TDB";

   TASKS ARE
      PERS_ADD_TASK              : TASK IS PERS_ADD_TASK;
      PERS_DISPLAY_TASK          : TASK IS PERS_DISPLAY_TASK;
      PERS_UPDATE_GENERAL_TASK   : TASK IS PERS_UPDATE_GENERAL_TASK;
      PERS_UPDATE_RAISEPRO_TASK  : TASK IS PERS_UPDATE_RAISEPRO_TASK;
      PERS_UPDATE_TRANSFER_TASK  : TASK IS PERS_UPDATE_TRANSFER_TASK;
      PERS_UPDATE_STATUS_TASK    : TASK IS PERS_UPDATE_STATUS_TASK;
   END TASKS;

   SERVER IS PERS_SERVER:
      PROCEDURE SERVER IMAGE IS "PERS$EXE:PERSONNEL.EXE";
      PROCEDURES ARE
         PERS_ADD, PERS_GET_DISPLAY, PERS_GET_EMPLOYEE,
         PERS_UPDATE_EMPLOYEE, PERS_GET_RAISEPRO, PERS_UPDATE_RAISEPRO,
         PERS_GET_TRANSFER, PERS_UPDATE_TRANSFER, PERS_GET_STATUS,
         PERS_UPDATE_STATUS;
      INITIALIZATION PROCEDURE IS PERS_STARTUP;
      TERMINATION PROCEDURE IS PERS_SHUTDOWN;
      DEFAULT OBJECT FILE IS "PERS$OBJ:PERSONNEL.OBJ";
   END SERVER;

END DEFINITION;
```

A.1.12 Message File

```
.TITLE  PERSMSG Messages for Personnel Application
.IDENT /Version 1.0/

.FACILITY PERSONNEL,21 /PREFIX=PRS_

.SEVERITY WARNING
DUPEMPNOS     <An employee with this number already exists.>
RECLOCK       <Record is locked by another user; press RETURN to try again.>
RECNOTFD      <Employee not found; try another number of exit.>

.SEVERITY FATAL
DBFAIL        <Database contains invalid data. Notify administrator.>
.END
```

A.1.13 Application Definition

```
REPLACE APPLICATION PERS_APPL

   AUDIT;
   APPLICATION USERNAME IS PERS$EXC;

   SERVER DEFAULTS ARE
      AUDIT;
      USERNAME IS PERS$SERVER;
      MAXIMUM SERVER PROCESSES IS 2;
      MINIMUM SERVER PROCESSES IS 2;
   END SERVER DEFAULTS;

   TASK DEFAULTS ARE
      AUDIT;
   END TASK DEFAULTS;

   TASK GROUP IS
      PERS_TASK_GROUP : TASK GROUP FILE IS "PERS$EXE:PERSGROUP.TDB";
   END TASK GROUP;

END DEFINITION;
```

A.1.14 Menu Definition

```
REPLACE MENU PERS_MENU

   HEADER IS "                          AVERTZ PERSONNEL SYSTEM";

   ENTRIES ARE
      ADD         : TASK IS PERS_ADD_TASK IN PERS_APPL;
                    TEXT IS "Add new employee";
      DISPLAY     : TASK IS PERS_DISPLAY_TASK IN PERS_APPL;
                    TEXT IS "Display current employee information";
      UPDATE_EMP  : TASK IS PERS_UPDATE_GENERAL_TASK IN PERS_APPL;
                    TEXT IS "Update general employee information";
      UPDATE_RSP  : TASK IS PERS_UPDATE_RAISEPRO_TASK IN PERS_APPL;
                    TEXT IS "Give raise and/or promotion";
      UPDATE_TRN  : TASK IS PERS_UPDATE_TRANSFER_TASK IN PERS_APPL;
                    TEXT IS "Record transfer to another department";
      UPDATE_STS  : TASK IS PERS_UPDATE_STATUS_TASK IN PERS_APPL;
                    TEXT IS "Change employee work status";
   END ENTRIES;

END DEFINITION;
```

A.2 AVERTZ Car Rental Application

The car rental application is built on a VAX DBMS database and includes three tasks. This section contains the complete sources for the application. Table A-1 lists each type of source definition, the sections of this appendix that contain them, and the specific task to which each definition applies.

Table A-2: Car Rental Application Sources

Object	Section	Related Task
Database Schema	A.2.1.1	All
Database Subschema	A.2.1.2	All
Database Storage Schema	A.2.1.3	All
Workspace	A.2.2	All
Task Definitions	A.2.3.1	Reservation Task
	A.2.4.1	Checkout Task
	A.2.5.1	Checkin Task
Form Definitions	A.2.3.2	Reservation Task
	A.2.4.2 A.2.4.3	Checkout Task
	A.2.5.2	Checkin Task
Request Definitions	A.2.3.3 A.2.3.4 A.2.3.5	Reservation Task
	A.2.4.4 A.2.4.5 A.2.4.6	Checkout Task
	A.2.5.3 A.2.5.4	Checkin Task
Step Procedures	A.2.3.6 A.2.3.7	Reservation Task

(continued on next page)

Table A-2: Car Rental Application Sources (Cont.)

Object	Section	Related Task
Step Procedures	A.2.3.6 A.2.3.7	Reservation Task
	A.2.4.7 A.2.4.8	Checkout Task
	A.2.5.5 A.2.5.6	Checkin Task
Server Procedures	A.2.6.1 A.2.6.2	All
Request Library Definition	A.2.7	All
Task Group Definition	A.2.8	All
Message Source File	A.2.9	All
Application Definition	A.2.10	All
Menu Definition	A.2.11	All

A.2.1 Car Rental Database Definition

A.2.1.1 Schema Definition

```
* CDD path to schema is "CDD$TOP.AVERTZ.AVERTZSC"
*-----------------------------------------------------------------------

SCHEMA NAME IS AVERTZSC

AREA NAME IS COMPANY_AREA

AREA NAME IS CUSTOMER_AREA

AREA NAME IS LOCATION_AREA

RECORD NAME IS COMPANY
   WITHIN COMPANY_AREA
      ITEM IS CO_NAME                 TYPE IS CHARACTER 25
      ITEM IS CO_DISCOUNT             TYPE IS SIGNED LONGWORD
      ITEM IS CO_ADDR_DATA_1          TYPE IS CHARACTER 25
      ITEM IS CO_ADDR_DATA_2          TYPE IS CHARACTER 25
      ITEM IS CO_CITY                 TYPE IS CHARACTER 20
      ITEM IS CO_STATE                TYPE IS CHARACTER 2
      ITEM IS CO_POSTAL_CODE          TYPE IS CHARACTER 9
      ITEM IS CO_PHONE                TYPE IS CHARACTER 10
      ITEM IS CO_CREDIT_CHECK         TYPE IS CHARACTER 2

RECORD NAME IS CUSTOMER
   WITHIN CUSTOMER_AREA
      ITEM IS CU_LAST_NAME            TYPE IS CHARACTER 20
      ITEM IS CU_FIRST_NAME           TYPE IS CHARACTER 15
      ITEM IS CU_INITIAL              TYPE IS CHARACTER 1
      ITEM IS CU_ADDR_DATA_1          TYPE IS CHARACTER 25
      ITEM IS CU_ADDR_DATA_2          TYPE IS CHARACTER 25
      ITEM IS CU_CITY                 TYPE IS CHARACTER 20
      ITEM IS CU_STATE                TYPE IS CHARACTER 2
      ITEM IS CU_POSTAL_CODE          TYPE IS CHARACTER 9
      ITEM IS CU_PHONE                TYPE IS CHARACTER 10
      ITEM IS CU_LICENSE_NO           TYPE IS CHARACTER 15
      ITEM IS CU_LICENSE_STATE        TYPE IS CHARACTER 2

RECORD NAME IS LOCATION
   WITHIN LOCATION_AREA
      ITEM IS LO_CODE                 TYPE IS CHARACTER 2
      ITEM IS LO_RES_NUM              TYPE IS SIGNED LONGWORD
      ITEM IS LO_NAME                 TYPE IS CHARACTER 25
      ITEM IS LO_ADDR_DATA_1          TYPE IS CHARACTER 25
      ITEM IS LO_ADDR_DATA_2          TYPE IS CHARACTER 25
      ITEM IS LO_CITY                 TYPE IS CHARACTER 20
      ITEM IS LO_STATE                TYPE IS CHARACTER 2
      ITEM IS LO_POSTAL_CODE          TYPE IS CHARACTER 9
      ITEM IS LO_PHONE                TYPE IS CHARACTER 10

RECORD NAME IS RESERVATION
   WITHIN CUSTOMER_AREA
      ITEM IS R_PICKUP_LOCATION       TYPE IS CHARACTER 2
      ITEM IS RESERVATION_NUM         TYPE IS SIGNED LONGWORD
      ITEM IS R_CAR_TYPE_CODE         TYPE IS SIGNED LONGWORD
      ITEM IS R_PICKUP_DATE           TYPE IS DATE
```

(continued on next page)

```
RECORD NAME IS CAR_TYPE
   WITHIN LOCATION_AREA
      ITEM IS CAR_TYPE_CODE                  TYPE IS SIGNED LONGWORD
      ITEM IS DAILY_RATE_LT_7_DAYS           TYPE IS SIGNED LONGWORD
      ITEM IS DAILY_RATE_GT_7_LT_30_DAYS     TYPE IS SIGNED LONGWORD
      ITEM IS DAILY_RATE_GT_30_DAYS          TYPE IS SIGNED LONGWORD
      ITEM IS DAILY_RATE_FUTURE_USE          TYPE IS SIGNED LONGWORD

RECORD NAME IS CAR
   WITHIN LOCATION_AREA
      ITEM IS CAR_NUM                  TYPE IS SIGNED LONGWORD
      ITEM IS CAR_TYPE_CODE            TYPE IS SIGNED LONGWORD
      ITEM IS CAR_MAKE                 TYPE IS CHARACTER 8
      ITEM IS CAR_YEAR                 TYPE IS CHARACTER 2
      ITEM IS LICENSE_NUM              TYPE IS CHARACTER 10
      ITEM IS LICENSE_STATE            TYPE IS CHARACTER 2

SET NAME IS COMPANY_CALC
   OWNER IS SYSTEM
   MEMBER IS COMPANY
      INSERTION IS AUTOMATIC RETENTION IS FIXED

SET NAME IS CUSTOMER_CALC
   OWNER IS SYSTEM
   MEMBER IS CUSTOMER
      INSERTION IS AUTOMATIC RETENTION IS FIXED

SET NAME IS LOCATION_CALC
   OWNER IS SYSTEM
   MEMBER IS LOCATION
      INSERTION IS AUTOMATIC RETENTION IS FIXED

SET NAME IS CUSTOMER_RESERVATION
   OWNER IS CUSTOMER
   MEMBER IS RESERVATION
      INSERTION IS AUTOMATIC RETENTION IS FIXED
      ORDER IS FIRST

SET NAME IS EMPLOYEE
   OWNER IS COMPANY
   MEMBER IS CUSTOMER
      INSERTION IS MANUAL RETENTION IS OPTIONAL
      ORDER IS LAST

SET NAME IS TYPE_AVAILABLE
   OWNER IS LOCATION
   MEMBER IS CAR_TYPE
      INSERTION IS AUTOMATIC RETENTION IS FIXED
      ORDER IS LAST

SET NAME IS CHECKED_IN_CARS
   OWNER IS CAR_TYPE
   MEMBER IS CAR
      INSERTION IS AUTOMATIC RETENTION IS OPTIONAL
      ORDER IS LAST
```

```
SET NAME IS CHECKED_OUT_CARS
   OWNER IS RESERVATION
   MEMBER IS CAR
      INSERTION IS MANUAL RETENTION IS OPTIONAL
      ORDER IS FIRST

SET NAME IS LOCATION_RESERVATION
   OWNER IS LOCATION
   MEMBER IS RESERVATION
      INSERTION IS AUTOMATIC RETENTION IS OPTIONAL
      ORDER IS FIRST
```

A.2.1.2 Subschema Definition

```
* CDD path to schema is "CDD$TOP.AVERTZ.AVERTZSC"
*-------------------------------------------------------------------------

SUBSCHEMA NAME IS AVERTZSS FOR AVERTZSC SCHEMA

REALM COMPANY_AREA
   IS COMPANY_AREA

REALM CUSTOMER_AREA
   IS CUSTOMER_AREA

REALM LOCATION_AREA
   IS LOCATION_AREA

RECORD NAME IS COMPANY
   ITEM IS CO_NAME                  TYPE IS CHARACTER 25
   ITEM IS CO_DISCOUNT              TYPE IS SIGNED LONGWORD
   ITEM IS CO_ADDR_DATA_1           TYPE IS CHARACTER 25
   ITEM IS CO_ADDR_DATA_2           TYPE IS CHARACTER 25
   ITEM IS CO_CITY                  TYPE IS CHARACTER 20
   ITEM IS CO_STATE                 TYPE IS CHARACTER 2
   ITEM IS CO_POSTAL_CODE           TYPE IS CHARACTER 9
   ITEM IS CO_PHONE                 TYPE IS CHARACTER 10
   ITEM IS CO_CREDIT_CHECK          TYPE IS CHARACTER 2

RECORD NAME IS CUSTOMER
   GROUP NAME IS CU_NAME
      ITEM IS CU_LAST_NAME          TYPE IS CHARACTER 20
      ITEM IS CU_FIRST_NAME         TYPE IS CHARACTER 15
      ITEM IS CU_INITIAL            TYPE IS CHARACTER 1
   ENDGROUP CU_NAME
   ITEM IS CU_ADDR_DATA_1           TYPE IS CHARACTER 25
   ITEM IS CU_ADDR_DATA_2           TYPE IS CHARACTER 25
   ITEM IS CU_CITY                  TYPE IS CHARACTER 20
   ITEM IS CU_STATE                 TYPE IS CHARACTER 2
   ITEM IS CU_POSTAL_CODE           TYPE IS CHARACTER 9
   ITEM IS CU_PHONE                 TYPE IS CHARACTER 10
   ITEM IS CU_LICENSE_NO            TYPE IS CHARACTER 15
   ITEM IS CU_LICENSE_STATE         TYPE IS CHARACTER 2
```

(continued on next page)

```
RECORD NAME IS LOCATION
   GROUP RESERVATION_ID
      ITEM IS LO_CODE                  TYPE IS CHARACTER 2
      ITEM IS LO_RES_NUM               TYPE IS SIGNED LONGWORD
   ENDGROUP RESERVATION_ID
   ITEM IS LO_NAME                     TYPE IS CHARACTER 25
   ITEM IS LO_ADDR_DATA_1              TYPE IS CHARACTER 25
   ITEM IS LO_ADDR_DATA_2              TYPE IS CHARACTER 25
   ITEM IS LO_CITY                     TYPE IS CHARACTER 20
   ITEM IS LO_STATE                    TYPE IS CHARACTER 2
   ITEM IS LO_POSTAL_CODE              TYPE IS CHARACTER 9
   ITEM IS LO_PHONE                    TYPE IS CHARACTER 10

RECORD NAME IS RESERVATION
   GROUP RESERVATION_ID
      ITEM IS R_PICKUP_LOCATION        TYPE IS CHARACTER 2
      ITEM IS RESERVATION_NUM          TYPE IS SIGNED LONGWORD
   ENDGROUP RESERVATION_ID
   ITEM IS R_CAR_TYPE_CODE             TYPE IS SIGNED LONGWORD
   ITEM IS R_PICKUP_DATE               TYPE IS DATE

RECORD NAME IS CAR_TYPE
   ITEM IS CAR_TYPE_CODE                    TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_LT_7_DAYS             TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_GT_7_LT_30_DAYS       TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_GT_30_DAYS            TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_FUTURE_USE            TYPE IS SIGNED LONGWORD

RECORD NAME IS CAR
   ITEM CAR_NUM                        TYPE IS SIGNED LONGWORD
   ITEM CAR_TYPE_CODE                  TYPE IS SIGNED LONGWORD
   ITEM CAR_MAKE                       TYPE IS CHARACTER 8
   ITEM CAR_YEAR                       TYPE IS CHARACTER 2
   ITEM LICENSE_NUM                    TYPE IS CHARACTER 10
   ITEM LICENSE_STATE                  TYPE IS CHARACTER 2

SET NAME IS COMPANY_CALC

SET NAME IS CUSTOMER_CALC

SET NAME IS LOCATION_CALC

SET NAME IS CUSTOMER_RESERVATION

SET NAME IS EMPLOYEE

SET NAME IS TYPE_AVAILABLE

SET NAME IS CHECKED_IN_CARS

SET NAME IS CHECKED_OUT_CARS

SET NAME IS LOCATION_RESERVATION
```

A.2.1.3 Storage Schema Definition

```
* CDD path to schema is "CDD$TOP.AVERTZ.AVERTZSC"
*-----------------------------------------------------------------------

STORAGE SCHEMA NAME IS AVERTZST FOR AVERTZSC SCHEMA

RECORD NAME IS COMPANY
   PLACEMENT IS CLUSTERED VIA COMPANY_CALC
   ITEM IS CO_NAME                 TYPE IS CHARACTER 25
   ITEM IS CO_DISCOUNT             TYPE IS SIGNED LONGWORD
   ITEM IS CO_ADDR_DATA_1          TYPE IS CHARACTER 25
   ITEM IS CO_ADDR_DATA_2          TYPE IS CHARACTER 25
   ITEM IS CO_CITY                 TYPE IS CHARACTER 20
   ITEM IS CO_STATE                TYPE IS CHARACTER 2
   ITEM IS CO_POSTAL_CODE          TYPE IS CHARACTER 9
   ITEM IS CO_PHONE                TYPE IS CHARACTER 10
   ITEM IS CO_CREDIT_CHECK         TYPE IS CHARACTER 2

RECORD NAME IS CUSTOMER
   PLACEMENT IS CLUSTERED VIA CUSTOMER_CALC
   ITEM IS CU_LAST_NAME            TYPE IS CHARACTER 20
   ITEM IS CU_FIRST_NAME           TYPE IS CHARACTER 15
   ITEM IS CU_INITIAL              TYPE IS CHARACTER 1
   ITEM IS CU_ADDR_DATA_1          TYPE IS CHARACTER 25
   ITEM IS CU_ADDR_DATA_2          TYPE IS CHARACTER 25
   ITEM IS CU_CITY                 TYPE IS CHARACTER 20
   ITEM IS CU_STATE                TYPE IS CHARACTER 2
   ITEM IS CU_POSTAL_CODE          TYPE IS CHARACTER 9
   ITEM IS CU_PHONE                TYPE IS CHARACTER 10
   ITEM IS CU_LICENSE_NO           TYPE IS CHARACTER 15
   ITEM IS CU_LICENSE_STATE        TYPE IS CHARACTER 2

RECORD NAME IS LOCATION
   PLACEMENT IS CLUSTERED VIA LOCATION_CALC
   ITEM IS LO_CODE                 TYPE IS CHARACTER 2
   ITEM IS LO_RES_NUM              TYPE IS SIGNED LONGWORD
   ITEM IS LO_NAME                 TYPE IS CHARACTER 25
   ITEM IS LO_ADDR_DATA_1          TYPE IS CHARACTER 25
   ITEM IS LO_ADDR_DATA_2          TYPE IS CHARACTER 25
   ITEM IS LO_CITY                 TYPE IS CHARACTER 20
   ITEM IS LO_STATE                TYPE IS CHARACTER 2
   ITEM IS LO_POSTAL_CODE          TYPE IS CHARACTER 9
   ITEM IS LO_PHONE                TYPE IS CHARACTER 10

RECORD NAME IS RESERVATION
   PLACEMENT IS CLUSTERED VIA CUSTOMER_RESERVATION
   ITEM IS R_PICKUP_LOCATION       TYPE IS CHARACTER 2
   ITEM IS RESERVATION_NUM         TYPE IS SIGNED LONGWORD
   ITEM IS R_CAR_TYPE_CODE         TYPE IS SIGNED LONGWORD
   ITEM IS R_PICKUP_DATE           TYPE IS DATE

RECORD NAME IS CAR_TYPE
   PLACEMENT IS CLUSTERED VIA TYPE_AVAILABLE
   ITEM IS CAR_TYPE_CODE               TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_LT_7_DAYS        TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_GT_7_LT_30_DAYS  TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_GT_30_DAYS       TYPE IS SIGNED LONGWORD
   ITEM IS DAILY_RATE_FUTURE_USE       TYPE IS SIGNED LONGWORD
```

(continued on next page)

```
RECORD NAME IS CAR
   PLACEMENT IS SCATTERED USING CAR_NUM
   ITEM CAR_NUM                    TYPE IS SIGNED LONGWORD
   ITEM CAR_MAKE                   TYPE IS CHARACTER 8
   ITEM CAR_YEAR                   TYPE IS CHARACTER 2
   ITEM LICENSE_NUM                TYPE IS CHARACTER 10
   ITEM LICENSE_STATE              TYPE IS CHARACTER 2

SET NAME IS COMPANY_CALC
   MODE IS CALC
      MEMBER IS COMPANY
         KEY IS CO_NAME

SET NAME IS CUSTOMER_CALC
   MODE IS CALC
      MEMBER IS CUSTOMER
         KEY IS CU_LAST_NAME
                CU_FIRST_NAME
                CU_INITIAL

SET NAME IS LOCATION_CALC
   MODE IS CALC
      MEMBER IS LOCATION
         KEY IS LO_CODE

SET NAME IS CUSTOMER_RESERVATION
   MODE IS CHAIN

SET NAME IS EMPLOYEE
   MODE IS CHAIN

SET NAME IS TYPE_AVAILABLE
   MODE IS CHAIN

SET NAME IS CHECKED_IN_CARS
   MODE IS CHAIN

SET NAME IS CHECKED_OUT_CARS
   MODE IS CHAIN

SET NAME IS LOCATION_RESERVATION
   MODE IS CHAIN
```

A.2.2 Workspace Definition

```
DEFINE RECORD AVERTZ_WORKSPACE
   DESCRIPTION IS
      /* Workspace to hold miscellaneous fields */.

AVERTZ_WORKSPACE STRUCTURE.
   PROGRAM_REQUEST_KEY          DATATYPE TEXT SIZE 6
                                INITIAL_VALUE IS "      ".
   RES_MADE                     DATATYPE TEXT SIZE 1
                                INITIAL_VALUE IS "N".
   TOTAL_OWED                   DATATYPE UNSIGNED NUMERIC SIZE 7
                                         SCALE -2.
   NUM_DAY                      DATATYPE SIGNED LONGWORD.
   DAYS_TO_CURRENT              DATATYPE SIGNED LONGWORD.
   DAYS_TO_RENTAL               DATATYPE SIGNED LONGWORD.
   DAYS_RENTED                  DATATYPE SIGNED LONGWORD.

END AVERTZ_WORKSPACE STRUCTURE.

END AVERTZ_WORKSPACE.
```

A.2.3 Definitions for the Reservation Task

A.2.3.1 AVERTZ_RESERVE_TASK Definition

```
REPLACE TASK AVERTZ_RESERVE_TASK

WORKSPACES ARE
   AVERTZ_WORKSPACE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;

BLOCK WORK
   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST1
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE,
         CAR_TYPE, LOCATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT RETRIEVAL"
      CALL AVERTZ_GET_RATES IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CAR_TYPE, LOCATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;
```

(continued on next page)

```
   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST2
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CAR_TYPE,
         COMPANY, CUSTOMER, LOCATION, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT"    : EXIT TASK;
         "REPEAT" : REPEAT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT UPDATE"
      CALL AVERTZ_RESERVE_CAR IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CAR_TYPE, COMPANY, CUSTOMER,
         LOCATION, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS AVERTZ_RESERVE_REQUEST3
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
         CUSTOMER, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
        "EXIT"    : EXIT TASK;
        "CHKOUT" : GOTO TASK AVERTZ_CHECKOUT_TASK
                   PASSING AVERTZ_WORKSPACE, COMPANY, CUSTOMER,
                   RESERVATION;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.2.3.2 AVERTZ_RESERVE_FORM Definition

```
                    R E S E R V A T I O N   F O R M


     Type of car:     99999                    Pickup location: AA
     Daily rate:     999.99
     Weekly rate:    999.99
     Monthly rate:   999.99

     Location name: XXXXXXXXXXXXXXXXXXX
     Address: XXXXXXXXXXXXXXXXXXXXX
              XXXXXXXXXXXXXXXXXXXXX
     City:    XXXXXXXXXXXXXXXXXX      State: AA  Postal code: XXXXXXXX
     Phone:   AAA-AAA-AAAA

     Company:  XXXXXXXXXXXXXXXXXX
     Customer: XXXXXXXXXXXXXXXXXX X XXXXXXXXXXXXXX

     Pickup date: 99-AAA-99

 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ZK-00056-00

A.2.3.3 AVERTZ_RESERVE_REQUEST1 Definition

```
REPLACE REQUEST AVERTZ_RESERVE_REQUEST1

   FORM IS CDD$TOP.AVERTZ.AVERTZ_RESERVE_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION;

   DESCRIPTION /* Accept car type code and location code */;

   USE FORM AVERTZ_RESERVE_FORM;
```

(continued on next page)

```
   INPUT CAR_TYPE_CODE TO CAR_TYPE_CODE,
         LO_CODE       TO LO_CODE;

   OUTPUT 'Enter car type code and pickup location code.' TO INFORM_LINE,
          'Press GOLD-E to exit from this task.' TO PRK_LINE;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.3.4 AVERTZ_RESERVE_REQUEST2 Definition

```
REPLACE REQUEST AVERTZ_RESERVE_REQUEST2

   FORM IS CDD$TOP.AVERTZ.AVERTZ_RESERVE_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVAT

   DESCRIPTION /* Display rates and location information as output;
                  accept company and customer names and pickup date
                  as input */;

   USE FORM AVERTZ_RESERVE_FORM;

   OUTPUT DAILY_RATE_LT_7_DAYS       TO DAY_RATE,
          DAILY_RATE_GT_7_LT_30_DAYS TO WEEK_RATE,
          DAILY_RATE_GT_30_DAYS      TO MONTH_RATE,
          LO_NAME                    TO LO_NAME,
          LO_ADDR_DATA_1             TO LO_ADDRESS1,
          LO_ADDR_DATA_2             TO LO_ADDRESS2,
          LO_CITY                    TO LO_CITY,
          LO_STATE                   TO LO_STATE,
          LO_POSTAL_CODE             TO LO_POSTAL_CODE,
          LO_PHONE                   TO LO_PHONE;
```

```
   OUTPUT 'Enter company (if any), customer, pickup date.'
             TO INFORM_LINE,
          'Press GOLD-E to exit, GOLD-R to repeat this task.'
             TO PRK_LINE;

   INPUT COMPANY     TO CO_NAME,
         FIRST_NAME  TO CU_FIRST_NAME,
         INITIAL     TO CU_INITIAL,
         LAST_NAME   TO CU_LAST_NAME,
         PICKUP_DATE TO R_PICKUP_DATE;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   PROGRAM KEY IS GOLD "R"
      NO CHECK;
      RETURN "REPEAT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.3.5 AVERTZ_RESERVE_REQUEST3 Definition

```
REPLACE REQUEST AVERTZ_RESERVE_REQUEST3

   FORM IS CDD$TOP.AVERTZ.AVERTZ_RESERVE_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;

   DESCRIPTION /* Inform the user that the reservation was
                  successfully entered in the database */;

   USE FORM AVERTZ_RESERVE_FORM;

   OUTPUT 'Reservation made.' TO INFORM_LINE,
          'Press GOLD-E to exit, GOLD-K to check out.' TO PRK_LINE;

   WAIT;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;
```

(continued on next page)

```
   PROGRAM KEY IS GOLD "K"
      NO CHECK;
      RETURN "CHKOUT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.3.6 AVERTZ_GET_RATES Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    AVERTZ_GET_RATES.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.

01 LOC-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_LOCNOTFD.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                 PIC S9(9) COMP
                              VALUE IS EXTERNAL DBM$_END.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE'
   FROM DICTIONARY
   REPLACING ==CAR_TYPE. == BY ==CAR_TYPE_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION'
   FROM DICTIONARY
   REPLACING ==LOCATION. == BY ==LOCATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CAR_TYPE_LINKAGE
                         LOCATION_LINKAGE
                  GIVING STATUS-RESULT.
```

```
MAIN SECTION.
010-FIND-LOCATION.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Find the pickup location and move location information to the
* linkage record for display.

    MOVE LO_CODE OF LOCATION_LINKAGE TO LO_CODE OF LOCATION.

    FETCH FIRST LOCATION WITHIN LOCATION_CALC
            USING LO_CODE OF LOCATION
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE LOCATION TO LOCATION_LINKAGE.

020-FIND-CAR-TYPE.

* Find the car type and move the rental rates to the linkage
* record for display.

    MOVE CAR_TYPE_CODE OF CAR_TYPE_LINKAGE TO CAR_TYPE_CODE OF CAR_TYPE.

    FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE
            USING CAR_TYPE_CODE OF CAR_TYPE
        ON ERROR
            PERFORM 052-ERROR-CHECK THRU 052-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

030-GET-RATES.
    MOVE CAR_TYPE TO CAR_TYPE_LINKAGE.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* If location is not found, display a message; signal any other errors

    IF DB-CONDITION EQUAL DBM$_END
       MOVE LOC-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.
```

(continued on next page)

```
052-ERROR-CHECK.
* If car type is not found, signal (form definition prevents user from
* entering a car type other than 01, 02, or 03)

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

052-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.3.7 AVERTZ_RESERVE_CAR Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    AVERTZ_RESERVE_CAR.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.

01 COM-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_COMNOTFD.
01 CREDIT-BAD               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_CREDITBD.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                 PIC S9(9) COMP
                              VALUE IS EXTERNAL DBM$_END.
01 DBM$_DUPNOTALL           PIC S9(9) COMP
                              VALUE IS EXTERNAL DBM$_DUPNOTALL.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE
   FROM DICTIONARY
   REPLACING ==CAR_TYPE. == BY ==CAR_TYPE_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY'
   FROM DICTIONARY
   REPLACING ==COMPANY. == BY ==COMPANY_LINKAGE. ==.
```

```
COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER"
   FROM DICTIONARY
   REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.LOCATION"
   FROM DICTIONARY
   REPLACING ==LOCATION. == BY ==LOCATION_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION"
   FROM DICTIONARY
   REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CAR_TYPE_LINKAGE
                         COMPANY_LINKAGE
                         CUSTOMER_LINKAGE
                         LOCATION_LINKAGE
                         RESERVATION_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
010-COMPANY-CHECK.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* See whether customer is using a corporate account.  If so,
* check that the company's credit is OK.  If the credit is not OK,
* issue a message and roll back.

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        PERFORM 015-CREDIT-CHECK THRU 015-CREDIT-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM
    ELSE
        GO TO 020-CUSTOMER-CHECK.

015-CREDIT-CHECK.
    MOVE CO_NAME OF COMPANY_LINKAGE TO CO_NAME OF COMPANY.

    FETCH FIRST COMPANY WITHIN COMPANY_CALC
            USING CO_NAME OF COMPANY
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 015-CREDIT-CHECK-EXIT.

    IF CO_CREDIT_CHECK OF COMPANY = "NO"
    THEN
        MOVE CREDIT-BAD TO STATUS-RESULT.

015-CREDIT-CHECK-EXIT.
    EXIT.
```

(continued on next page)

```
020-CUSTOMER-CHECK.

* See whether customer is on file.  If not, add new customer
* information. If the new customer is an employee of a company
* on file, connect the customer to the company.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            PERFORM 025-NEW-CUSTOMER THRU 025-NEW-CUSTOMER-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    MOVE CUSTOMER TO CUSTOMER_LINKAGE.

    GO TO 040-STORE-RESERVATION.

025-NEW-CUSTOMER.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        PERFORM 028-ADD-CUSTOMER THRU 028-ADD-CUSTOMER-EXIT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        CONNECT CUSTOMER TO EMPLOYEE.

025-NEW-CUSTOMER-EXIT.
    EXIT.

028-ADD-CUSTOMER.
    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.
    MOVE SPACES TO CU_ADDR_DATA_1 OF CUSTOMER.
    MOVE SPACES TO CU_ADDR_DATA_2 OF CUSTOMER.
    MOVE SPACES TO CU_CITY OF CUSTOMER.
    MOVE SPACES TO CU_STATE OF CUSTOMER.
    MOVE SPACES TO CU_POSTAL_CODE OF CUSTOMER.
    MOVE SPACES TO CU_PHONE OF CUSTOMER.
    MOVE SPACES TO CU_LICENSE_NO OF CUSTOMER.
    MOVE SPACES TO CU_LICENSE_STATE OF CUSTOMER.

    STORE CUSTOMER
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

028-ADD-CUSTOMER-EXIT.
    EXIT.
```

```
040-STORE-RESERVATION.

* Move reservation information into the reservation record for
* display and store the reservation under the customer and under
* the requested pickup location.

    SET STATUS-RESULT TO SUCCESS.

    MOVE LO_CODE OF LOCATION_LINKAGE TO LO_CODE OF LOCATION,
        R_PICKUP_LOCATION OF RESERVATION.

    FETCH FIRST LOCATION WITHIN LOCATION_CALC USING
            LO_CODE OF LOCATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    ADD 1 TO LO_RES_NUM OF LOCATION.

    MOVE LO_RES_NUM OF LOCATION TO RESERVATION_NUM OF RESERVATION.

    MODIFY LO_RES_NUM OF LOCATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE CAR_TYPE_CODE OF CAR_TYPE_LINKAGE TO R_CAR_TYPE_CODE
        OF RESERVATION.

    MOVE R_PICKUP_DATE OF RESERVATION_LINKAGE TO R_PICKUP_DATE
        OF RESERVATION.

    STORE RESERVATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE 'Y' TO RES_MADE OF AVERTZ_WORKSPACE.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* If company not found, display an error message; signal any other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE COM-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

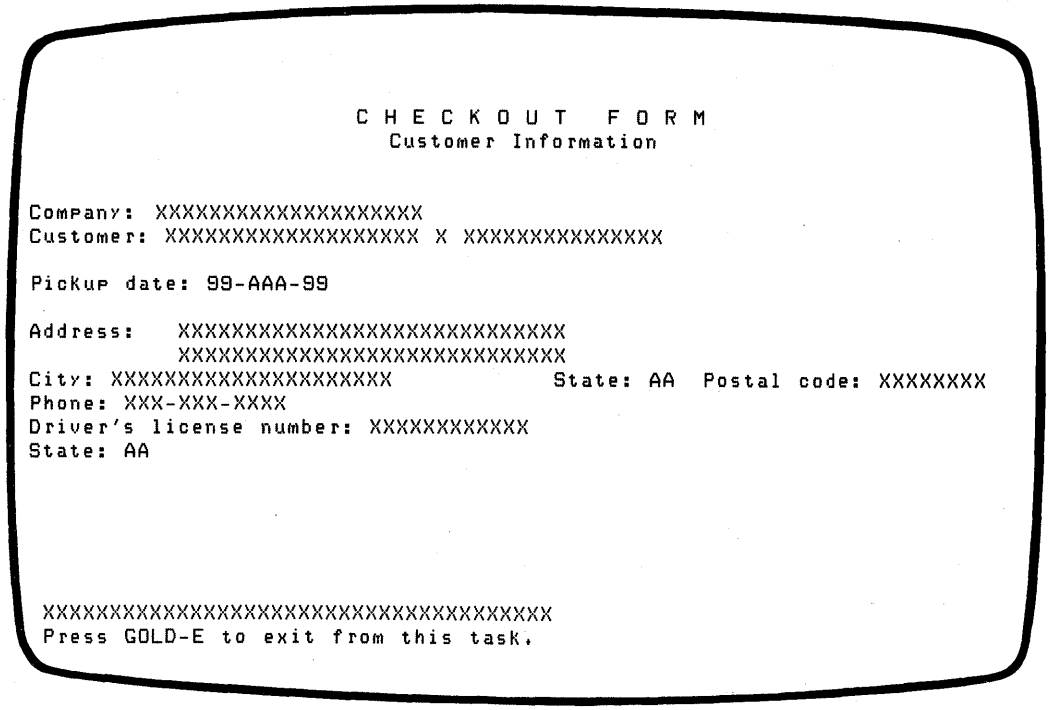
A.2.4 Definitions for the Checkout Task

A.2.4.1 AVERTZ_CHECKOUT_TASK Definition

```
REPLACE TASK AVERTZ_CHECKOUT_TASK

WORKSPACES ARE
   AVERTZ_WORKSPACE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;

BLOCK WORK
   EXCHANGE
      NO EXCHANGE;
      CONTROL FIELD IS RES_MADE
         "Y" : GOTO STEP CHECK_CUSTOMER;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS AVERTZ_CHECKOUT_REQUEST1
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
         CUSTOMER, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT RETRIEVAL"
      CALL AVERTZ_FIND_RESERVATION IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, COMPANY, CUSTOMER, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

CHECK_CUSTOMER:
   EXCHANGE
      REQUEST IS AVERTZ_CHECKOUT_REQUEST2
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
         CUSTOMER, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT UPDATE"
      CALL AVERTZ_ASSIGN_CAR IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CAR, CUSTOMER, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO STEP CHECK_CUSTOMER;
      END CONTROL FIELD;
```

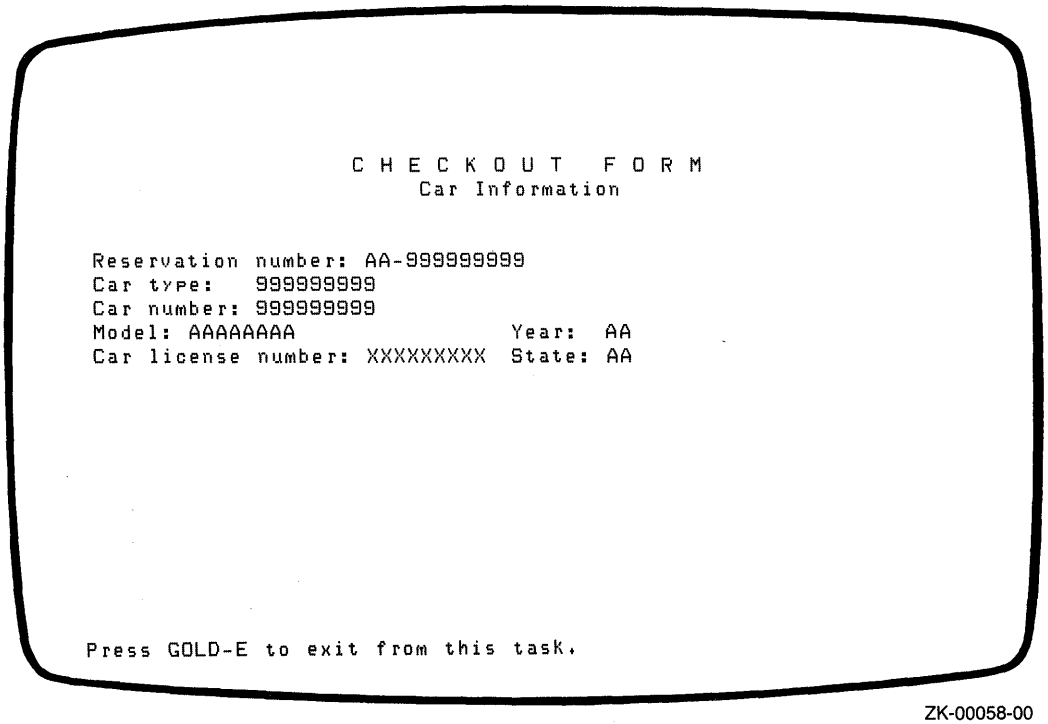
```
    EXCHANGE
       REQUEST IS AVERTZ_CHECKOUT_REQUEST3
          USING AVERTZ_WORKSPACE, CAR, RESERVATION;
       CONTROL FIELD IS PROGRAM_REQUEST_KEY
          "EXIT" : EXIT TASK;
       END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.2.4.2 AVERTZ_CHECKOUT_FORM1 Definition



ZK-00057-00

A.2.4.3 AVERTZ_CHECKOUT_FORM2 Definition



ZK-00058-00

A.2.4.4 AVERTZ_CHECKOUT_REQUEST1 Definition

```
REPLACE REQUEST AVERTZ_CHECKOUT_REQUEST1

   FORM IS CDD$TOP.AVERTZ.AVERTZ_CHECKOUT_FORM1;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVAT
```

```
   DESCRIPTION /* Accept company and customer names as input
                  for retrieving a reservation */;

   USE FORM AVERTZ_CHECKOUT_FORM1;

   OUTPUT 'Enter company (if any) and customer name.   ' TO INFORM_LINE;

   INPUT COMPANY     TO CO_NAME,
         FIRST_NAME  TO CU_FIRST_NAME,
         INITIAL     TO CU_INITIAL,
         LAST_NAME   TO CU_LAST_NAME;

   RETURN PICKUP_DATE TO R_PICKUP_DATE;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.4.5 AVERTZ_CHECKOUT_REQUEST2 Definition

```
REPLACE REQUEST AVERTZ_CHECKOUT_REQUEST2

   FORM IS CDD$TOP.AVERTZ.AVERTZ_CHECKOUT_FORM1;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;

   DESCRIPTION /* Display customer data (if customer is on file) and
                  accept changes (or new information, if new customer */;

   USE FORM AVERTZ_CHECKOUT_FORM1;

   OUTPUT 'Enter or change customer data if necessary.'
       TO INFORM_LINE;

   OUTPUT CO_NAME          TO COMPANY,
          CU_FIRST_NAME    TO FIRST_NAME,
          CU_INITIAL       TO INITIAL,
          CU_LAST_NAME     TO LAST_NAME,
          CU_ADDR_DATA_1   TO ADDRESS1,
          CU_ADDR_DATA_2   TO ADDRESS2,
          CU_CITY          TO CITY,
          CU_STATE         TO STATE,
          CU_POSTAL_CODE   TO POSTAL_CODE,
```

(continued on next page)

```
        CU_PHONE         TO PHONE,
        CU_LICENSE_NO    TO LICENSE_NUMBER,
        CU_LICENSE_STATE TO LICENSE_STATE;

   INPUT ADDRESS1        TO CU_ADDR_DATA_1,
         ADDRESS2        TO CU_ADDR_DATA_2,
         CITY            TO CU_CITY,
         STATE           TO CU_STATE,
         POSTAL_CODE     TO CU_POSTAL_CODE,
         PHONE           TO CU_PHONE,
         LICENSE_NUMBER TO CU_LICENSE_NO,
         LICENSE_STATE  TO CU_LICENSE_STATE;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   END DEFINITION;
```

A.2.4.6 AVERTZ_CHECKOUT_REQUEST3 Definition

```
REPLACE REQUEST AVERTZ_CHECKOUT_REQUEST3

   FORM IS CDD$TOP.AVERTZ.AVERTZ_CHECKOUT_FORM2;

   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVAT

   DESCRIPTION /* Display information about the car checked out
                  to a customer */;

   USE FORM AVERTZ_CHECKOUT_FORM2;

   OUTPUT R_PICKUP_LOCATION TO LO_CODE,
          RESERVATION_NUM   TO RES_NUMBER,
          CAR_TYPE_CODE     TO CAR_TYPE_CODE,
          CAR_NUM           TO CAR_NUMBER,
          CAR_MAKE          TO CAR_MODEL,
          CAR_YEAR          TO CAR_YEAR,
          LICENSE_NUM       TO CAR_LICENSE,
          LICENSE_STATE     TO CAR_LICENSE_STATE;

   WAIT;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

END DEFINITION;
```

A.2.4.7 AVERTZ_FIND_RESERVATION Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.     AVERTZ_FIND_RESERVATION.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.
01 COM-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_COMNOTFD.
01 CREDIT-BAD               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_CREDITBD.
01 CUS-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_CUSNOTFD.
01 RES-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_RESNOTFD.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                 PIC S9(9) COMP
                              VALUE IS EXTERNAL DBM$_END.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY"
   FROM DICTIONARY
   REPLACING ==COMPANY. == BY ==COMPANY_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER"
   FROM DICTIONARY
   REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION"
   FROM DICTIONARY
   REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         COMPANY_LINKAGE
                         CUSTOMER_LINKAGE
                         RESERVATION_LINKAGE
                 GIVING STATUS-RESULT.

MAIN SECTION.
010-GET-CUSTOMER-INFO.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

(continued on next page)

```
* If customer is renting the car for a company, check the
* company's credit.  If the credit is not OK, roll back.

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        PERFORM 040-COMPANY-CHECK THRU 040-COMPANY-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM
    ELSE
        GO TO 030-FIND-RESERVATION.


030-FIND-RESERVATION.

* Find customer's reservation and move the customer record so it
* can be displayed.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            PERFORM 052-ERROR-CHECK THRU 052-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE R_PICKUP_DATE OF RESERVATION_LINKAGE TO R_PICKUP_DATE
        OF RESERVATION.

    FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION
            USING R_PICKUP_DATE OF RESERVATION
        ON ERROR
            PERFORM 054-ERROR-CHECK THRU 054-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE CUSTOMER TO CUSTOMER_LINKAGE.

    GO TO 100-EXIT-PROGRAM.

040-COMPANY-CHECK.
    MOVE CO_NAME OF COMPANY_LINKAGE TO CO_NAME OF COMPANY.

    FETCH FIRST COMPANY WITHIN COMPANY_CALC
            USING CO_NAME OF COMPANY
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 040-COMPANY-CHECK-EXIT.
```

```
    IF CO_CREDIT_CHECK OF COMPANY = "NO"
    THEN
        MOVE CREDIT-BAD TO STATUS-RESULT.

040-COMPANY-CHECK-EXIT.
    EXIT.

050-ERROR-CHECK.
* If company is not found, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE COM-NOT-FOUND TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.

052-ERROR-CHECK.
* If customer is not found, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE CUS-NOT-FOUND TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

052-ERROR-CHECK-EXIT.
    EXIT.

054-ERROR-CHECK.
* If reservation is not found, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE RES-NOT-FOUND TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

054-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

(continued on next page)

A.2.4.8 AVERTZ_ASSIGN_CAR Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    AVERTZ_ASSIGN_CAR.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.

01 RECORD-LOCKED              PIC S9(9) COMP
                               VALUE IS EXTERNAL AVZ_RECLOCK.
01 OUT-OF-CARS                PIC S9(9) COMP
                               VALUE IS EXTERNAL AVZ_NOMORCAR.
01 DB-FAILURE                 PIC S9(9) COMP
                               VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_DEADLOCK              PIC S9(9) COMP
                               VALUE IS EXTERNAL DBM$_DEADLOCK.
01 DBM$_END                   PIC S9(9) COMP
                               VALUE IS EXTERNAL DBM$_END.
01 STATUS-RESULT              PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR"
   FROM DICTIONARY
   REPLACING ==CAR. == BY ==CAR_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER"
   FROM DICTIONARY
   REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVAT]
   FROM DICTIONARY
   REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CAR_LINKAGE
                         CUSTOMER_LINKAGE
                         RESERVATION_LINKAGE
                   GIVING STATUS-RESULT.

MAIN SECTION.
010-UPDATE-CUSTOMER.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.
```

```
* Find the customer and modify the customer record if necessary;
* any error is fatal.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE CUSTOMER_LINKAGE TO CUSTOMER.

    MODIFY CUSTOMER
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

* Find the customer's reservation (use the pickup date to distinguish,
* as customer may have several reservations).  Any error is fatal.

020-FIND-RESERVATION.
    MOVE R_PICKUP_DATE OF RESERVATION_LINKAGE TO R_PICKUP_DATE
        OF RESERVATION.

    FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION
            USING R_PICKUP_DATE OF RESERVATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE RESERVATION_ID OF RESERVATION TO RESERVATION_ID
        OF RESERVATION_LINKAGE.

* Find an available car at the pickup location.  Move the car from the
* checked-in set to the checked-out set under the correct reservation.
* All errors are fatal.

030-ASSIGN-CAR.
    FIND OWNER WITHIN LOCATION_RESERVATION.

    MOVE R_CAR_TYPE_CODE OF RESERVATION TO CAR_TYPE_CODE OF CAR_TYPE.

    FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE
            USING CAR_TYPE_CODE OF CAR_TYPE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".
```

(continued on next page)

```
    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 060-OUT-OF-CARS THRU 060-OUT-OF-CARS-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE CAR TO CAR_LINKAGE.

    DISCONNECT FROM CHECKED_IN_CARS.

    CONNECT TO CHECKED_OUT_CARS.

    GO TO 100-EXIT-PROGRAM.

050-ERROR-CHECK.
* If customer record is locked, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_DEADLOCK
    THEN
        MOVE RECORD-LOCKED TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.

060-OUT-OF-CARS.
* If location is out of cars of the requested type, find a car
* of another size.

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       EVALUATE CAR_TYPE_CODE OF CAR_TYPE
          WHEN 1 PERFORM 070-OUT-OF-COMPACTS THRU 070-OUT-OF-COMPACTS-EXIT
          WHEN 2 PERFORM 080-OUT-OF-MIDSIZE THRU 080-OUT-OF-MIDSIZE-EXIT
          WHEN 3 PERFORM 090-OUT-OF-FULLSIZE THRU 090-OUT-OF-FULLSIZE-EXIT
       END-EVALUATE
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

060-OUT-OF-CARS-EXIT.
    EXIT.

070-OUT-OF-COMPACTS.

* Since all compact cars are checked out, try midsize cars; if midsize
* cars are all checked out, try fullsize cars.  If those are gone,
* location is completely out of cars.
```

```
    FIND NEXT CAR_TYPE WITHIN TYPE_AVAILABLE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 072-ERROR-CHECK THRU 072-ERROR-CHECK-EXIT.

070-OUT-OF-COMPACTS-EXIT.
    EXIT.

072-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        FIND NEXT CAR_TYPE WITHIN TYPE_AVAILABLE
            ON ERROR
                MOVE DB-FAILURE TO STATUS-RESULT
                CALL "DBM$SIGNAL"
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 074-ERROR-CHECK THRU 074-ERROR-CHECK-EXIT.

072-ERROR-CHECK-EXIT.
    EXIT.

074-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE OUT-OF-CARS TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

074-ERROR-CHECK-EXIT.
    EXIT.

080-OUT-OF-MIDSIZE.

* Since all midsize cars are checked out, try fullsize cars;
* if fullsize cars are all checked out, try compact cars.  If
* those are gone, location is completely out of cars.

    FIND NEXT CAR_TYPE WITHIN TYPE_AVAILABLE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 082-ERROR-CHECK THRU 082-ERROR-CHECK-EXIT.

080-OUT-OF-MIDSIZE-EXIT.
    EXIT.
```

(continued on next page)

```
082-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        FIND PRIOR CAR_TYPE WITHIN TYPE_AVAILABLE
            ON ERROR
                MOVE DB-FAILURE TO STATUS-RESULT
                CALL "DBM$SIGNAL"
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
             PERFORM 084-ERROR-CHECK THRU 084-ERROR-CHECK-EXIT.

082-ERROR-CHECK-EXIT.
    EXIT.

084-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE OUT-OF-CARS TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

084-ERROR-CHECK-EXIT.
    EXIT.

090-OUT-OF-FULLSIZE.

* Since all fullsize cars are checked out, try midsize cars; if
* midsize cars are all checked out, try compact cars.  If those
* are gone, location is completely out of cars.

    FIND PRIOR CAR_TYPE WITHIN TYPE_AVAILABLE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 092-ERROR-CHECK THRU 092-ERROR-CHECK-EXIT.

090-OUT-OF-FULLSIZE-EXIT.
    EXIT.

092-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        FIND PRIOR CAR_TYPE WITHIN TYPE_AVAILABLE
            ON ERROR
                MOVE DB-FAILURE TO STATUS-RESULT
                CALL "DBM$SIGNAL"
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".
```

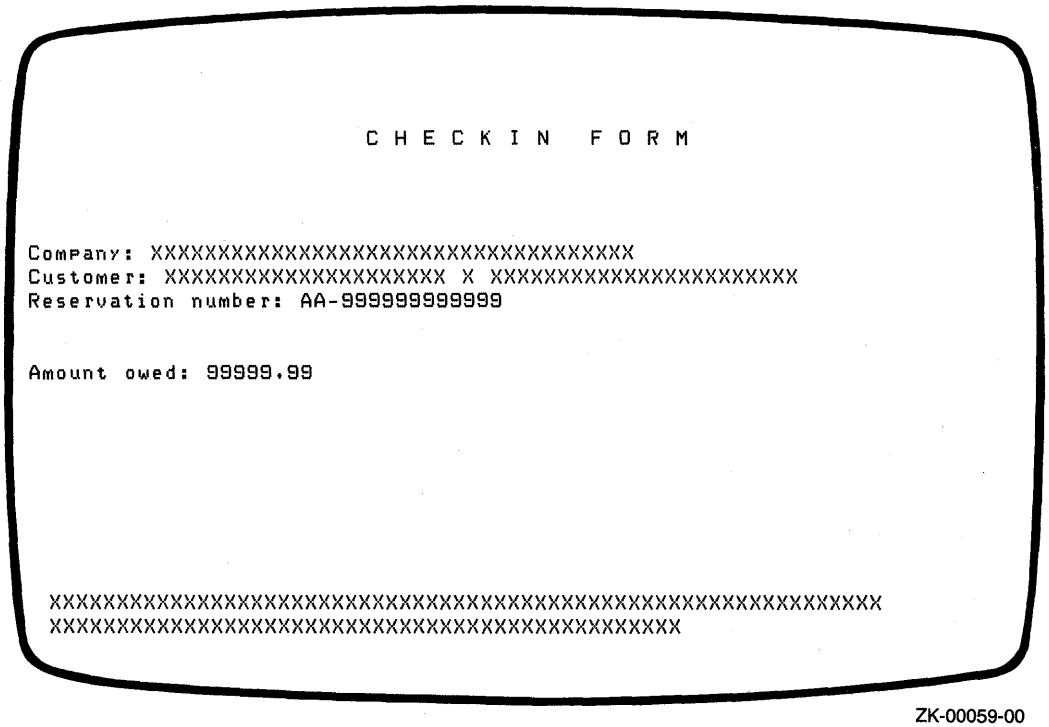
```
    FETCH FIRST CAR WITHIN CHECKED_IN_CARS
        ON ERROR
            PERFORM 094-ERROR-CHECK THRU 094-ERROR-CHECK-EXIT.

092-ERROR-CHECK-EXIT.
    EXIT.

094-ERROR-CHECK.
    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE OUT-OF-CARS TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

094-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.5 Definitions for the Checkin Task

A.2.5.1 AVERTZ_CHECKIN_TASK Definition

```
REPLACE TASK AVERTZ_CHECKIN_TASK

WORKSPACES ARE
   AVERTZ_WORKSPACE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CAR_TYPE,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER,
   CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION

BLOCK WORK
   EXCHANGE
      REQUEST IS AVERTZ_CHECKIN_REQUEST1
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, COMPANY,
         CUSTOMER, RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT RETRIEVAL"
      CALL AVERTZ_CHECKIN IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, COMPANY, CUSTOMER, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               GOTO PREVIOUS EXCHANGE;
      END CONTROL FIELD;

   EXCHANGE
      REQUEST IS AVERTZ_CHECKIN_REQUEST2
         USING ACMS$PROCESSING_STATUS, AVERTZ_WORKSPACE, CUSTOMER,
         RESERVATION;
      CONTROL FIELD IS PROGRAM_REQUEST_KEY
         "EXIT" : EXIT TASK;
      END CONTROL FIELD;

   PROCESSING WITH DBMS RECOVERY "READY CONCURRENT UPDATE"
      CALL AVERTZ_RETURN_CAR IN AVERTZ_SERVER
         USING AVERTZ_WORKSPACE, CUSTOMER, RESERVATION;
      CONTROL FIELD IS ACMS$T_STATUS_TYPE
         "B" : GET ERROR MESSAGE;
               ROLLBACK;
               REPEAT TASK;
      END CONTROL FIELD;

END BLOCK WORK;

END DEFINITION;
```

A.2.5.2 AVERTZ_CHECKIN_FORM Definition



ZK-00059-00

A.2.5.3 AVERTZ_CHECKIN_REQUEST1 Definition

```
REPLACE REQUEST AVERTZ_CHECKIN_REQUEST1

   FORM IS CDD$TOP.AVERTZ.AVERTZ_CHECKIN_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION;
```

(continued on next page)

```
   DESCRIPTION /* Accept company and customer names and a
                  reservation number for a returned car */;

   USE FORM AVERTZ_CHECKIN_FORM;

   OUTPUT 'Enter company (if any), customer name, and reservation number.'
             TO INFORM_LINE,
          'Press GOLD-E to exit from this task.' TO PRK_LINE;

   INPUT COMPANY          TO CO_NAME,
         FIRST_NAME       TO CU_FIRST_NAME,
         INITIAL          TO CU_INITIAL,
         LAST_NAME        TO CU_LAST_NAME,
         LO_CODE          TO R_PICKUP_LOCATION,
         RES_NUMBER       TO RESERVATION_NUM;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;

   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.5.4 AVERTZ_CHECKIN_REQUEST2 Definition

```
REPLACE REQUEST AVERTZ_CHECKIN_REQUEST2

   FORM IS CDD$TOP.AVERTZ.AVERTZ_CHECKIN_FORM;

   RECORD IS
      CDD$TOP.ACMS$DIR.ACMS$WORKSPACES.ACMS$PROCESSING_STATUS;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZ_WORKSPACE;
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER
   RECORD IS
      CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVAT

   DESCRIPTION /* Display the amount owed by the customer */;

   USE FORM AVERTZ_CHECKIN_FORM;

   OUTPUT ' ' TO INFORM_LINE,
          'Press GOLD-E to exit, RETURN to continue.' TO PRK_LINE;
   OUTPUT TOTAL_OWED TO TOTAL_OWED;

   WAIT;

   PROGRAM KEY IS GOLD "E"
      NO CHECK;
      RETURN "EXIT" TO PROGRAM_REQUEST_KEY;
   END PROGRAM KEY;
```

```
   CONTROL FIELD IS ACMS$T_STATUS_TYPE
      "B" : MESSAGE LINE IS ACMS$T_STATUS_MESSAGE;
   END CONTROL FIELD;

   END DEFINITION;
```

A.2.5.5 AVERTZ_CHECKIN Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    AVERTZ_CHECKIN.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.
01 CUS-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_CUSNOTFD.
01 RES-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_RESNOTFD.
01 COM-NOT-FOUND            PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_COMNOTFD.
01 DB-FAILURE               PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_DBFAIL.
01 DBM$_END                 PIC S9(9) COMP
                              VALUE IS EXTERNAL DBM$_END.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE"
   FROM DICTIONARY
   REPLACING ==AVERTZ_WORKSPACE. == BY ==AVERTZ_WORKSPACE_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.COMPANY"
   FROM DICTIONARY
   REPLACING ==COMPANY. == BY ==COMPANY_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER"
   FROM DICTIONARY
   REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION"
   FROM DICTIONARY
   REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE_LINKAGE
                         COMPANY_LINKAGE
                         CUSTOMER_LINKAGE
                         RESERVATION_LINKAGE
                   GIVING STATUS-RESULT.
```

(continued on next page)

```
MAIN SECTION.
010-FIND-RESERVATION.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Find the customer and the reservation.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            PERFORM 050-ERROR-CHECK THRU 050-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

    MOVE RESERVATION_ID OF RESERVATION_LINKAGE TO RESERVATION_ID
        OF RESERVATION.

    FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION
            USING RESERVATION_ID OF RESERVATION
        ON ERROR
            PERFORM 052-ERROR-CHECK THRU 052-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 100-EXIT-PROGRAM.

* Find the rates based on the car type.

    FETCH FIRST CAR WITHIN CHECKED-OUT-CARS
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

* See whether you gave the customer the size of car he asked for.  If
* he got a bigger car than he requested, charge him the rates for the
* type requested.  If he got a smaller car, charge him the rates for
* the type he got.

    IF R_CAR_TYPE_CODE OF RESERVATION GREATER THAN CAR_TYPE_CODE OF CAR
    THEN
        MOVE CAR_TYPE_CODE OF CAR TO CAR_TYPE_CODE OF CAR_TYPE
    ELSE
        MOVE R_CAR_TYPE_CODE OF RESERVATION TO CAR_TYPE_CODE OF CAR_TYPE.

    FIND OWNER WITHIN LOCATION_RESERVATION.

    FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE USING
            CAR_TYPE_CODE OF CAR_TYPE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".
```

```
020-COMPUTE-CHARGES.

* Call LIB$DAY to get the number of days between 17 November, 1858,
* and the current date.

    CALL "LIB$DAY" USING BY REFERENCE NUM-DAY
                         BY VALUE 0
                   GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        CALL "LIB$SIGNAL" USING STATUS-RESULT.

    MOVE NUM-DAY TO DAYS_TO_CURRENT.

* Call LIB$DAY to get the number of days between 17 November, 1858,
* and the rental date.

    CALL "LIB$DAY" USING BY REFERENCE NUM-DAY
                                      R_PICKUP_DATE OF RESERVATION
                   GIVING STATUS-RESULT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        CALL "LIB$SIGNAL" USING STATUS-RESULT.

    MOVE NUM-DAY TO DAYS_TO_RENTAL.

    SUBTRACT DAYS_TO_RENTAL FROM DAYS_TO_CURRENT GIVING DAYS_RENTED.

    IF DAYS_RENTED = 0
    THEN
        ADD 1 TO DAYS_RENTED.

    IF DAYS_RENTED GREATER THAN 30
    THEN
        MULTIPLY DAILY_RATE_GT_30_DAYS OF CAR_TYPE
        BY DAYS_RENTED GIVING TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE
    ELSE
        IF DAYS_RENTED LESS THAN 7
        THEN
            MULTIPLY DAILY_RATE_LT_7_DAYS OF CAR_TYPE
            BY DAYS_RENTED GIVING TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE
        ELSE
            MULTIPLY DAILY_RATE_GT_7_LT_30_DAYS OF CAR_TYPE
            BY DAYS_RENTED GIVING TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE
        END-IF
    END-IF.

* See whether this was a business rental and apply the corporate
* discount, if any.

    IF CO_NAME OF COMPANY_LINKAGE NOT EQUAL SPACES
    THEN
        PERFORM 054-COMPANY-DISCOUNT THRU 054-COMPANY-DISCOUNT-EXIT.

    GO TO 100-EXIT-PROGRAM.
```

(continued on next page)

```
050-ERROR-CHECK.
* If customer is not found, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE CUS-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

050-ERROR-CHECK-EXIT.
    EXIT.

052-ERROR-CHECK.
* If reservation is not found, return an error message; signal any other
* errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
       MOVE RES-NOT-FOUND TO STATUS-RESULT
    ELSE
       MOVE DB-FAILURE TO STATUS-RESULT
       CALL "DBM$SIGNAL".

052-ERROR-CHECK-EXIT.
    EXIT.

054-COMPANY-DISCOUNT.
* If this is a corporate rental, check for company discount and apply
* to rates.

    MOVE CO_NAME OF COMPANY_LINKAGE TO CO_NAME OF COMPANY.

    FETCH FIRST COMPANY WITHIN COMPANY_CALC
          USING CO_NAME OF COMPANY
        ON ERROR
          PERFORM 056-ERROR-CHECK THRU 056-ERROR-CHECK-EXIT.

    IF STATUS-RESULT NOT SUCCESS
    THEN
        GO TO 054-COMPANY-DISCOUNT-EXIT.

    IF CO_DISCOUNT OF COMPANY NOT = ZEROS
    THEN
        COMPUTE TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE =
            TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE -
                (TOTAL_OWED OF AVERTZ_WORKSPACE_LINKAGE *
                     (CO_DISCOUNT OF COMPANY / 100))
    END-IF.

054-COMPANY-DISCOUNT-EXIT.
    EXIT.
```

```
056-ERROR-CHECK.
* If company is not found, return an error message; signal any
* other errors

    IF DB-CONDITION EQUAL DBM$_END
    THEN
        MOVE COM-NOT-FOUND TO STATUS-RESULT
    ELSE
        MOVE DB-FAILURE TO STATUS-RESULT
        CALL "DBM$SIGNAL".

056-ERROR-CHECK-EXIT.
    EXIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.5.6 AVERTZ_RETURN_CAR Procedure

```
IDENTIFICATION DIVISION.

PROGRAM-ID.    AVERTZ_RETURN_CAR.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.

DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.
01 DB-FAILURE               PIC S9(9) COMP
                             VALUE IS EXTERNAL AVZ_DBFAIL.
01 STATUS-RESULT            PIC S9(9) COMP.

LINKAGE SECTION.
COPY "CDD$TOP.AVERTZ.AVERTZ_WORKSPACE" FROM DICTIONARY.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.CUSTOMER"
   FROM DICTIONARY
   REPLACING ==CUSTOMER. == BY ==CUSTOMER_LINKAGE. ==.

COPY "CDD$TOP.AVERTZ.AVERTZSC.DBM$SUBSCHEMAS.AVERTZSS.DBM$RECORDS.RESERVATION"
   FROM DICTIONARY
   REPLACING ==RESERVATION. == BY ==RESERVATION_LINKAGE. ==.

PROCEDURE DIVISION USING AVERTZ_WORKSPACE
                         CUSTOMER_LINKAGE
                         RESERVATION_LINKAGE
                 GIVING STATUS_RESULT.
```

(continued on next page)

```
MAIN SECTION.
010-RETURN-CAR.

    SET STATUS-RESULT TO SUCCESS.

    INITIALIZE PROGRAM_REQUEST_KEY.

* Find customer; any error is fatal.

    MOVE CU_NAME OF CUSTOMER_LINKAGE TO CU_NAME OF CUSTOMER.

    FETCH FIRST CUSTOMER WITHIN CUSTOMER_CALC USING
            CU_NAME OF CUSTOMER
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

* Find reservation; any error is fatal.

    MOVE RESERVATION_ID OF RESERVATION_LINKAGE TO RESERVATION_ID
        OF RESERVATION.

    FETCH FIRST RESERVATION WITHIN CUSTOMER_RESERVATION USING
            RESERVATION_ID OF RESERVATION
        ON ERROR
              MOVE DB-FAILURE TO STATUS-RESULT
              CALL "DBM$SIGNAL".

* Find the car, the location, and the car type; disconnect the car
* from the checked-out set and connect it back to the checked-in set.
* Any errors are fatal.

    FETCH FIRST CAR WITHIN CHECKED_OUT_CARS
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE R_PICKUP_LOCATION OF RESERVATION TO LO_CODE OF LOCATION.

    FETCH FIRST LOCATION WITHIN LOCATION_CALC USING
            LO_CODE OF LOCATION
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    MOVE CAR_TYPE_CODE OF CAR TO CAR_TYPE_CODE OF CAR_TYPE.

    FETCH FIRST CAR_TYPE WITHIN TYPE_AVAILABLE USING
            CAR_TYPE_CODE OF CAR_TYPE
        ON ERROR
            MOVE DB-FAILURE TO STATUS-RESULT
            CALL "DBM$SIGNAL".

    FIND CURRENT CAR.

    DISCONNECT FROM CHECKED_OUT_CARS.

    CONNECT TO CHECKED_IN_CARS.
```

```
* Find customer's reservation and disconnect it.

    FIND CURRENT RESERVATION.

    DISCONNECT RESERVATION FROM LOCATION_RESERVATION.

    GO TO 100-EXIT-PROGRAM.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.6 Server Procedures

A.2.6.1 Initialization Procedure

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    AVERTZ_STARTUP.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.
DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.
01  DB-FAILURE              PIC S9(9) COMP
                              VALUE IS EXTERNAL AVZ_DBFAIL.
01  STATUS-RESULT           PIC S9(9) COMP.

PROCEDURE DIVISION GIVING STATUS-RESULT.
DECLARATIVES.
DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION.
FILE-CHECKING.
    MOVE DB-FAILURE TO STATUS-RESULT
    CALL "DBM$SIGNAL".
END DECLARATIVES.

MAIN SECTION.
000-START.
    SET STATUS-RESULT TO SUCCESS.
    READY CONCURRENT UPDATE.
    COMMIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.6.2 Termination Procedure

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    AVERTZ_SHUTDOWN.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

DATA DIVISION.
SUB-SCHEMA SECTION.
DB  AVERTZSS WITHIN AVERTZSC FOR "AVERTZ$APPL:AVERTZSC.ROO".

WORKING-STORAGE SECTION.
01  DB-FAILURE                  PIC S9(9) COMP
                                  VALUE IS EXTERNAL AVZ_DBFAIL.
01  STATUS-RESULT               PIC S9(9) COMP.

PROCEDURE DIVISION GIVING STATUS-RESULT.
DECLARATIVES.
DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION.
FILE-CHECKING.
    MOVE DB-FAILURE TO STATUS-RESULT
    CALL "DBM$SIGNAL".
END DECLARATIVES.

MAIN SECTION.
000-START.
    SET STATUS-RESULT TO SUCCESS.
    READY CONCURRENT UPDATE.
    COMMIT.

100-EXIT-PROGRAM.
    EXIT PROGRAM.
```

A.2.7 Request Library Definition

```
REPLACE LIBRARY AVERTZ_REQLIB

REQUEST IS AVERTZ_RESERVE_REQUEST1;
REQUEST IS AVERTZ_RESERVE_REQUEST2;
REQUEST IS AVERTZ_RESERVE_REQUEST3;
REQUEST IS AVERTZ_CHECKOUT_REQUEST1;
REQUEST IS AVERTZ_CHECKOUT_REQUEST2;
REQUEST IS AVERTZ_CHECKOUT_REQUEST3;
REQUEST IS AVERTZ_CHECKIN_REQUEST1;
REQUEST IS AVERTZ_CHECKIN_REQUEST2;

END DEFINITION;
```

A.2.8 Task Group Definition

```
REPLACE GROUP AVERTZ_TASK_GROUP

   REQUEST LIBRARY IS "AVERTZ$APPL:AVERTZ_REQLIB.RLB";

   MESSAGE FILE IS "AVERTZ$APPL:AVZMSG.EXE";

   TASKS ARE
      AVERTZ_RESERVE_TASK   : TASK IS AVERTZ_RESERVE_TASK;
      AVERTZ_CHECKOUT_TASK  : TASK IS AVERTZ_CHECKOUT_TASK;
      AVERTZ_CHECKIN_TASK   : TASK IS AVERTZ_CHECKIN_TASK;
   END TASKS;

   SERVER IS AVERTZ_SERVER:
      PROCEDURE SERVER IMAGE IS "AVERTZ$APPL:AVERTZ.EXE";
      PROCEDURES ARE
         AVERTZ_GET_RATES, AVERTZ_RESERVE_CAR, AVERTZ_FIND_RESERVATION,
         AVERTZ_ASSIGN_CAR, AVERTZ_CHECKIN, AVERTZ_RETURN_CAR;
      INITIALIZATION PROCEDURE IS AVERTZ_STARTUP;
      TERMINATION PROCEDURE IS AVERTZ_SHUTDOWN;
      DEFAULT OBJECT FILE IS "AVERTZ$OBJ:AVERTZ.OBJ";
   END SERVER;

END DEFINITION;
```

A.2.9 Message File

```
.TITLE  AVERTZMSG Messages for AVERTZ Application
.IDENT /Version 1.0/

.FACILITY AVERTZ,13 /PREFIX=AVZ_

.SEVERITY WARNING
COMNOTFD <No company with this name is an AVERTZ customer; please try again>
CREDITBD <Company's credit rating is bad; credit denied>
CUSNOTFD <No customer with this name is in the database; please try again>
LOCNOTFD <No AVERTZ location has this location code; please try again>
NOMORCAR <This location completely out of cars; notify management>
RECLOCK  <Record locked by another user; please try again>
RESNOTFD <No reservation was found for this customer>

.SEVERITY FATAL
DBFAIL   <Database contains invalid data. Notify administrator.>
.END
```

A.2.10 Application Definition

```
REPLACE APPLICATION AVERTZ_APPL

   AUDIT;
   APPLICATION USERNAME IS AVZ$EXC;

   SERVER DEFAULTS ARE
      AUDIT;
      USERNAME IS AVZ$SERVER;
      MAXIMUM SERVER PROCESSES IS 2;
      MINIMUM SERVER PROCESSES IS 0;
   END SERVER DEFAULTS;

   TASK DEFAULTS ARE
      AUDIT;
   END TASK DEFAULTS;

   TASK GROUP IS
      AVERTZ_TASK_GROUP : TASK GROUP FILE IS
                          "AVERTZ$APPL:AVERTZ_TASK_GROUP.TDB";
   END TASK GROUP;

END DEFINITION;
```

A.2.11 Menu Definition

```
REPLACE MENU AVERTZ_MENU

   HEADER IS "                                AVERTZ CAR RENTAL SYSTEM";

   ENTRIES ARE
      RESERVE  : TASK IS AVERTZ_RESERVE_TASK IN AVERTZ_APPL;
                 TEXT IS "Make Reservation";
      CHECKOUT : TASK IS AVERTZ_CHECKOUT_TASK IN AVERTZ_APPL;
                 TEXT IS "Check Out Car";
      CHECKIN  : TASK IS AVERTZ_CHECKIN_TASK IN AVERTZ_APPL;
                 TEXT IS "Check In Car";
   END ENTRIES;

END DEFINITION;
```

Index

In this index, a page number followed by "t" indicates a table reference. A page number followed by "f" indicates a figure reference.

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Introduction to Application Development with the VAX Information Architecture
AA-GR93A-TE

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

------Do Not Tear - Fold Here and Tape------

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: DISG Documentation ZKO2–2/N53
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, N.H. 03062

------Do Not Tear - Fold Here and Tape------

VAX Information Architecture Documentation Directory, Master Glossary, and Master Index

---

December 1985

This manual describes the documentation available for the VAX Information Architecture family of software products. It also includes a master glossary and a master index to the documentation sets.

OPERATING SYSTEM: **VMS**

MicroVMS

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

ACMS	DECUS	UNIBUS
CDD	MicroVAX	VAX
DATATRIEVE	MicroVMS	VAXcluster
DEC	PDP	VAX Information Architecture
DECgraph	Rdb/ELN	VMS
DECnet	Rdb/VMS	VT
DECslide	TDMS	digital™

Contents

How to Use This Manual

This manual describes the documentation of the VAX Information Architecture.

Intended Audience

This book is intended for all users of VAX Information Architecture products.

Operating System Information

To verify which versions of your operating system are compatible with these versions of the VAX Information Architecture products, check the most recent copy of the following:

- For the VMS operating system -- *VAX/VMS Optional Software Cross Reference Table*, SPD 25.99.xx
- For the MicroVMS operating system -- *MicroVMS Optional Software Cross Reference Table*, SPD 28.99.xx

Structure

There are three parts to this manual:

Chapter 1	Describes the documentation available for the VAX Information Architecture products.
Chapter 2	Is the master glossary of VAX Information Architecture terms.
Chapter 3	Is the master index to VAX Information Architecture documentation.

Note that the products described are often referred to by an abbreviated name. For example, the VAX DATATRIEVE software is referred to as DATATRIEVE, and the VAX TDMS software is referred to as TDMS.

Documentation Directory

Documentation Directory 1

Chapter 1 of this manual describes the documentation for the VAX Information Architecture products. There are three manuals that describe the VAX Information Architecture as a whole; they are grouped together in a binder entitled "Using the VAX Information Architecture." If you are just getting acquainted with the VAX Information Architecture, you should begin with these manuals. They are:

VAX Information Architecture Summary Description

Audience: All users

Content: This manual summarizes the features of each of the VAX Information Architecture products and explains how they work together.

VAX Information Architecture Documentation Directory, Master Glossary, and Master Index

Audience: All users

Content: This guide briefly describes the manuals in each of the documentation sets of the VAX Information Architecture components. It includes a master glossary of terms and a master index to all of the documentation sets.

Introduction to Application Development with the VAX Information Architecture

Audience: All users

Content: This manual uses examples to show you how to create application programs using the VAX Information Architecture products.

The following sections describe the documentation for each of the VAX Information Architecture products.

VAX Common Data Dictionary Version 3

VAX Common Data Dictionary Release Notes

Audience: All users

Content: This manual includes specific information about this CDD release as well as information included late in the development of this release.

VAX Common Data Dictionary Installation Guide

Audience: System managers/programmers

Content: This manual tells you how to install the CDD software and run the Installation Verification Procedure (IVP).

VAX Common Data Dictionary User's Guide

Audience: Programmers/data administrators/system managers

Content: This manual is a task-oriented guide to the use of the CDD and its three utilities: DMU, CDDV, and CDDL. In addition, it contains information useful to the data administrator or system manager about designing, creating, protecting, and maintaining the CDD.

VAX Common Data Dictionary Utilities Reference Manual

Audience: Programmers/data administrators

Content: This manual explains how to use the Dictionary Management Utility (DMU) and the Dictionary Verify/Fix Utility (CDDV).

VAX Common Data Dictionary Data Definition Language Reference Manual

Audience: Programmers/data administrators

Content: This manual provides a complete description of the Common Data Dictionary Data Definition Language Utility (CDDL). It also contains information about CDDL compatibility with VAX programming languages and VAX Information Architecture products.

VAX Rdb/VMS Version 2

VAX Rdb/VMS Release Notes

Audience: All users

Content: This manual includes specific information about this Rdb/VMS release as well as information included late in the development of this release.

VAX Rdb/VMS Guide to Data Manipulation

Audience: All users

Content: This guide shows how to retrieve, modify, and delete data. Use it as a tutorial to learn RDO, the interactive utility. The guide shows how to write queries that retrieve the desired information from the database.

VAX Rdb/VMS Reference Manual

Audience: All users

Content: This manual contains complete reference information on the components of the Rdb/VMS language, including RDO:

- Language elements, such as value expressions and record selection expressions
- Data definition statements
- Data manipulation statements
- Database maintenance utility statements
- Statements for setting up the interactive environment

It also includes a summary of the Rdb/VMS language arranged by function.

VAX Rdb/VMS Pocket Guide

Audience: All users

Content: This book summarizes the information in the *VAX Rdb/VMS Reference Manual.*

VAX Rdb/VMS Installation Guide

Audience: System managers

Content: This guide explains how to install Rdb/VMS.

VAX Rdb/VMS Guide to Programming

Audience: Programmers

Content: This manual shows how to write programs that use Rdb/VMS as a data access method. It includes information on writing programs in high-level languages that are supported by Rdb/VMS precompilers. It also includes a section on using callable RDO, a utility for languages without precompilers.

VAX Rdb/VMS Guide to Database Design and Definition

Audience: Database designers and administrators

Content: This manual explains how to design a database and how to set up definitions of database entities. It explains how you can design your database so that it is compatible with the features of VAX Rdb/VMS. The manual guides you from the analysis of your organization's information needs, through the process of designing logical and physical databases, to the use of the DEFINE statements to create the database.

VAX Rdb/VMS Guide to Database Maintenance and Administration

Audience: Database administrators and operators

Content: This guide shows how to use the database maintenance utilities to keep the database running and to keep its data consistent. It explains how to do backup and recovery, how to handle database journaling, and how to optimize the database's use of system resources.

VAX DBMS Version 3

VAX DBMS Release Notes

Audience: All users

Content: This manual includes specific information about this DBMS release as well as information included late in the development of this release.

VAX DBMS Programming Pocket Guide

Audience: All users

Content: This guide lists the syntax for the DBQ data manipulation language (DML) and the DML command, which invokes the DML precompiler.

VAX DBMS Installation Guide

Audience: System managers/database operators

Content: This guide explains the VAX DBMS installation procedure and describes the VAX/VMS parameters you can reset to optimize performance.

VAX DBMS Introduction to Data Manipulation

Audience: Application programmers

Content: This manual introduces the VAX DBMS data manipulation language (DML) and the Database Query (DBQ) utility. The manual shows how to locate, retrieve, store, and modify records in a database and explains the roles of set relationships and currency indicators in those tasks.

VAX DBMS Programming Guide

Audience: Application programmers

Content: This manual takes you step-by-step through the construction of various programs to illustrate how to use the features of VAX DBMS DML and the generic precompiler. The appendix shows sample bill-of-materials programs.

VAX DBMS Programming Reference Manual

Audience: Application programmers

Content: This manual describes the syntax and use of the VAX DBMS data manipulation language (DML) and the DML precompiler, the interactive use of the Database Query (DBQ) utility, and the use of the callable system subroutines. The appendixes list the DML and Database Control System (DBCS) monitor error messages, explanations, and appropriate user actions.

VAX DBMS FDML Reference Manual

Audience: FORTRAN application programmers

Content: This manual describes the FORTRAN Data Manipulation Language (FDML) syntax and semantics. It also shows how to use FDML statements in your programs and how to compile, link, and run those programs. The appendixes list the FDML error messages, explanations, and appropriate user actions.

VAX DBMS FDML Pocket Guide

Audience: FORTRAN application programmers

Content: This guide lists the syntax for the FORTRAN data manipulation language (FDML).

VAX DBMS Introduction to Database Administration

Audience: Database administrators

Content: This manual introduces the role of database administrator and basic database management concepts. It explains the purpose of database structures (record types, data item types, and set types) and data manipulation functions. This manual provides a general introduction to the VAX DBMS data definition language (DDL), with guidelines for producing schemas, storage schemas, subschemas, and security schemas. In addition, it introduces the VAX DBMS Database Operator (DBO) utility, which helps you create and maintain your database.

VAX DBMS Database Design Guide

Audience: Database administrators

Content: This guide shows an experienced database administrator how to design a VAX DBMS database once a production-level model has been developed. It explains:

- Defining logical and physical database attributes
- Developing user views
- Controlling VAX DBMS parameters that affect memory management
- Using the snapshot and space area management capabilities to your advantage

The appendix contains the schema, storage schema, and subschemas from the PARTS database, a sample database used throughout the VAX DBMS documentation.

VAX DBMS Database Security Guide

Audience: Database administrators

Content: This guide describes the VAX DBMS facilities that help you ensure the security of your database. This manual explains how to write and compile a security schema definition, how to assign security schemas to users, and how to control the use of DBO commands that can retrieve or modify the contents of your database. The appendix contains security schema definitions from the PARTS database.

VAX DBMS Database Maintenance and Performance Guide

Audience: Database administrators

Content: This guide is divided into two parts: the first describes maintenance activities you should regularly perform, while the second describes how to evaluate and improve performance. In addition, this guide shows you how to ensure the integrity of your database. This manual explains how to use the:

- DBO/ANALYZE command and DBO/SHOW commands to monitor and evaluate your database
- DBO/MODIFY command, DDL compiler commands, and the DBALTER facility to change many of the logical and physical characteristics of your database
- Database backup and journaling facilities to recover and restore your database if it has been corrupted
- DBO/VERIFY and DBO/DUMP commands to monitor data integrity
- DBALTER to make low-level changes to the database

This manual also describes how to use VAX DBMS in a VAXcluster environment.

VAX DBMS Database Load/Unload Guide

Audience: Database administrators

Content: This guide describes how to use the VAX DBMS load and unload facilities to perform an initial database load, extract data from your database, unload and reload your database, and physically restructure your database.

The appendix contains format and sequence language examples for each type of operation.

VAX DBMS Database Administration Reference Manual

Audience: Database administrators/database operators

Content: This manual describes the syntax and use of the data definition language (DDL) used to write the schema, storage schema, subschema, and security schema definitions and shows how to use the DDL compiler to compile, generate, and modify those definitions. It also describes the syntax for the Load Format Language, the Load Sequence Language, and the Unload Sequence Language definitions. It describes the syntax of each DDL definition, DDL compiler command, and DBO utility comand. This manual also describes the commands of the DBALTER facility, which is a low-level patch capability for VAX DBMS databases. The appendixes list the compile-time and DBO error and warning messages.

VAX DBMS Database Administration Pocket Guide

Audience: Database administrators

Content: This guide lists the syntax for the DBO commands, the DBALTER commands, the Load/Unload facility language, the data definition language, and the DDL compilation utility. It also lists the DDL reserved words.

VAX TDMS Version 1

VAX TDMS Release Notes

Audience: All users

Content: This manual includes specific information about this TDMS release as well as information included late in the development of this release.

VAX TDMS Sample Application Manual

Audience: All users

Content: This manual provides commented listings of the form definitions, record definitions, requests, and program code for each of the three sample applications provided with the VAX TDMS software.

VAX TDMS Quick Reference Guide

Audience: All users

Content: This manual provides listings, tables, and charts for the most frequently used TDMS features.

VAX TDMS Installation Guide

Audience: System managers

Content: This manual describes the VAX TDMS installation and verification procedures.

VAX TDMS Forms Manual

Audience: Form and application designers

Content: This manual describes how to create and use forms in a VAX TDMS application to provide information for or collect information from the terminal operator. In addition, this manual explains how to use the Form Definition Utility to create, modify, and store form definitions.

VAX TDMS Request Manual

Audience: Programmers/application designers

Content: This manual describes how to create and use requests in VAX TDMS applications to control the information displayed on the terminal and collected from the terminal operator. The manual also shows how to use the Request Definition Utility to create, modify, and store requests and request libraries in the CDD, and to build request library files.

VAX TDMS Application Programming Manual

Audience: Programmers/application designers

Content: This manual describes a program in a VAX TDMS application and discusses the relationships between the program and requests. It also explains how to use VAX TDMS programming calls within an application program and how to use the debugging facility provided by VAX TDMS. The manual includes programming examples in VAX COBOL and VAX BASIC. The syntax for VAX TDMS programming calls is shown in VAX COBOL, VAX BASIC, and VAX FORTRAN.

VAX TDMS V1.4 Documentation Supplement

Audience: Programmers/application designers

Content: This manual describes the VAX TDMS asynchronous programming calls, which are meant for advanced VAX TDMS applications. It also describes new features and performance enhancements available in V1.4 and later versions of VAX TDMS.

VAX ACMS Version 2

VAX ACMS Release Notes

Audience: All users

Content: This manual includes specific information about this ACMS release as well as information included late in the development of this release.

VAX ACMS Installation Guide

Audience: System managers

Content: This manual describes how to install VAX ACMS. It also describes how to run the installation verification procedures.

VAX ACMS Application Definition Guide

Audience: Application designers/managers

Content: This manual explains how to use the ACMS Application Definition Utility to define task groups, applications, and menus. It also explains how to modify the default menu format provided by ACMS.

VAX ACMS Application Definition Reference Manual

Audience: Application designers/managers/programmers

Content: This manual describes all syntax and functions of the Application Definition Utility.

VAX ACMS Application Management Guide

Audience: System/application managers, ACMS operators

Content: This manual explains how to authorize user and device access to ACMS, control ACMS applications, monitor ACMS system performance, and run the ACMS sample applications. A reference section describes the syntax and functions of the ACMS application management tools.

VAX ACMS Terminal User's Guide

Audience: Terminal operators

Content: This manual explains how to enter and exit from ACMS and how to select and control tasks from menus.

VAX ACMS Definition Pocket Guide

Audience: Application designers/managers/programmers

Content: This pocket guide lists the syntax you use with the VAX ACMS Application Definition Utility.

VAX ACMS Management Pocket Guide

Audience: System/application managers and ACMS operators

Content: This pocket guide lists the syntax for the VAX ACMS Operator commands, Audit Trail Report commands, Device Definition Utility commands, User Definition Utility commands, and ACMSGEN Utility commands.

VAX ACMS Application Design Guide

Audience: Application designers/programmers

Content: This manual provides a detailed explanation of the critical problems in designing ACMS applications. It describes how to address those problems using the ACMS tools. It also discusses other VMS tools you can use to help solve application design problems.

VAX ACMS Application Programming Guide

Audience: Application designers/programmers

Content: This manual explains how to write and debug programs for ACMS applications. It also provides reference information for the ACMS programming tools.

VAX ACMS Task Definition Guide

Audience: Application designers/programmers

Content: This manual explains how to create task and task group definitions using the Application Definition Utility.

Developing Applications with VAX ACMS

Audience: Application designers/programmers

Content: This manual uses a simple application and a step-by-step approach to explain how to develop a complete ACMS application.

VAX ACMS Demonstration Facility Card

Audience: Application designers/programmers

Content: This card explains how to run the ACMS Demonstration Facility (ADF).

VAX ACMS Programming Pocket Guide

Audience: Application designers/programmers

Content: This pocket guide lists the syntax for the VAX ACMS Task Debugger commands and application programming service.

VAX ACMS Systems Interface Programming Guide

Audience: System designers/programmers

Content: The ACMS Systems Interface is a group of services that an experienced systems programmer can use to access ACMS components from outside the standard ACMS environment. The guide describes the ACMS Systems Interface and explains the interface services a systems programmer can use to submit tasks to an ACMS system. It also explains how to allow communication between task submitters and their tasks.

VAX DATATRIEVE Version 3

VAX DATATRIEVE Release Notes

Audience: All users

Content: This manual includes specific information about this DATATRIEVE release as well as information included late in the development of this release.

VAX DATATRIEVE Installation Guide

Audience: System managers

Content: This manual describes the installation procedure for VAX DATATRIEVE.

VAX DATATRIEVE Handbook

Audience: Beginning and intermediate users

Content: This manual describes how to create VAX DATATRIEVE applications. It includes some tutorial information on describing data and creating procedures.

VAX DATATRIEVE Guide to Using Graphics

Audience: Intermediate or experienced users

Content: This manual introduces the use of VAX DATATRIEVE graphics and contains the reference material for creating DATATRIEVE plots.

VAX DATATRIEVE Guide to Writing Reports

Audience: Intermediate or experienced users

Content: This manual explains how to use the VAX DATATRIEVE Report Writer.

VAX DATATRIEVE User's Guide

Audience: Experienced users

Content: This manual contains information about the interactive use of VAX DATATRIEVE. It includes information on using DATATRIEVE to manipulate data and on its use with forms and database management systems. It also includes information on improving performance and working with remote data.

VAX DATATRIEVE Reference Manual

Audience: Experienced users

Content: This manual contains reference information for VAX DATATRIEVE.

VAX DATATRIEVE Guide to Programming and Customizing

Audience: Programmers/system managers

Content: This manual explains how to use the VAX DATATRIEVE Call Interface. It also describes how to create user-defined keywords and user-defined functions to customize DATATRIEVE and how to customize DATATRIEVE help and message texts.

VAX DATATRIEVE Pocket Guide

Audience: Experienced users

Content: This guide contains quick-reference information about using VAX DATATRIEVE.

Master Glossary

Master Glossary 2

access control list (ACL)

A table that lists which users are allowed access to an object and the kind of access they are allowed. The CDD maintains ACLs for DATATRIEVE and TDMS. ACMS, VAX DBMS, and Rdb/VMS maintain their own ACLs.

See also privilege.

access mode

A characteristic of a transaction that describes what kind of operation you intend to perform on data in a database.

ACL

See access control list.

ACMS

See Application Control and Management System.

ACMS Central Controller

The ACMS process that serves as the central control point for the ACMS run-time system.

ACMS Operator

An ACMS user authorized to control the daily operations of ACMS and/or its components with the ACMS Operator commands.

ACMS Operator command

One of a number of DCL commands provided by ACMS to control the operations of the ACMS system software and ACMS applications. Many ACMS Operator commands require the VMS OPER privilege.

active form

The TDMS form, referred to in a request, that is used during a single programming request call. A conditional request can refer to more than one form, but only one form can be active at any one time.

ADB

See application database.

ADT

See Application Design Tool.

ADU

See Application Definition Utility.

after-image journal

A file that contains copies of records after they have been updated. You can use the after-image journal to reconstruct a restored database up to the last successfully completed transaction. After-image journaling is also called long-term journaling.

agent

A VMS process through which one or more ACMS task submitters access the ACMS run-time system. All ACMS users submit tasks through an agent. ACMS provides one agent, called the command process, that acts for all task submitters.

See also command process.

AIJ

See after-image journal.

allow mode

A characteristic of a transaction that describes the type of access to data that you allow other users.

ancestor

A preceding dictionary or subdictionary directory in the CDD hierarchy. Ancestors have as descendants all related dictionary directories and objects below them in the hierarchy.

application

- A logically related set of data processing operations that support a particular business activity.
- In ACMS, a set of tasks that are related by the business activity they support and that are controlled as a single unit. An ACMS application is defined with the ACMS Application Definition Utility (ADU) and runs under the control of the ACMS run-time system. An application definition specifies operational characteristics for the tasks and servers of the task groups that make up the application.

Application Control and Management System (ACMS)

A software product, layered on VMS, used to define, run, and control online applications. You can use ACMS to control existing applications and applications developed with ACMS. It provides a task implementation method that uses high-level definitions to replace complex application code. These definitions reduce the programming time and maintenance costs of traditional application programs that do comparable work.

application database (ADB)

In ACMS, a database accessed at run-time that contains information derived from application and task group definitions. An application database is generated by building an application definition with the Application Definition Utility (ADU). The ACMS run-time system uses application databases to determine which processes to start, when to start them, and which users have access to which tasks.

Application Definition Utility (ADU)

The primary tool for creating ACMS applications. The Application Definition Utility provides the commands and clauses for defining tasks, task groups, applications, and menus.

Application Design Tool (ADT)

A DATATRIEVE utility that aids you in creating domains, record definitions, and files. For users who have not created record definitions or data files before, ADT provides a fast way to create a database by prompting with questions at each step in the process.

application execution controller (EXC)

The ACMS component that controls task execution for all the tasks in an application. Each application has its own application execution controller. Application execution controllers start up and control the server processes needed to handle processing work for tasks. They also handle exchange steps, step actions, and the sequencing of steps for tasks defined with ACMS. Application execution controllers refer to application databases, task databases, request libraries, and message files.

application program

A sequence of instructions and routines, not part of the basic operating system, designed to serve the specific needs of a user. An application program can use a database system to access data.

See also run unit.

application specification

A specification for an ACMS application that can consist of an application name, a logical name, a node name and file name, or a logical name and file name. You use application specifications in ADU clauses to identify applications on single-node or multiple-node (distributed) ACMS systems.

area

In VAX DBMS, a subdivision of the database, named in the schema, that corresponds to an RMS file. An area file, also known as a storage file, contains the data stored in that portion of the VAX DBMS database. Any number of record

types can be stored within an area. One or more areas make up a subschema realm.

See also realm.

array

A data structure consisting of more than one element, in which all elements have the same data types and are referred to by the same variable name.

In a TDMS form or record, an array is a field that contains several elements referred to by the same name in a request and that have the same characteristics (length, data type, and so on).

ascending order

An order of sorting that starts with the lowest value of a key and proceeds to the highest value, in accordance with the rules for comparing data items.

See also sort key, descending order.

asynchronous call

A call to a TDMS subroutine that begins, but does not necessarily complete, the requested operation before letting your program continue execution. At some later time, the requested operation finishes and notifies the program.

See also synchronous call.

attribute

See field attribute.

audit trail

In ACMS, a monitoring tool that has a recording facility and a utility for generating reports. The recording facility gathers information about a running ACMS system, including information about system and application starts and stops, user logins and logouts, processing errors, user task selections, task completions, and task cancellations. The report utility generates summary reports of this information.

In CDD, a collection of the history list entries for a dictionary directory, subdictionary, or object, created with the /AUDIT qualifier.

See also history list.

AUTOMATIC member

In VAX DBMS, a record that is automatically inserted into a specified set when the record is stored in the database. AUTOMATIC set membership is specified in the schema definition.

See also MANUAL member.

Bachman diagram

A graphic representation of the set relationships between owner and member records that is used to analyze and document a DBMS database design.

The VAX DBMS Database Query (DBQ) Utility displays Bachman diagrams.

background text

The text on a TDMS form that is displayed whenever the form is displayed. Background text often includes the names of fields, the name of the form, and instructions to the user.

bar chart

A chart that uses rectangular bars to compare values in fields or expressions. The height of the bars is proportional to the size of the fields or expressions represented.

See also line graph, pie chart, and scattergraph.

batch processing

A mode of computer operation in which the commands and data that control the actions of the computer are entered by a programmed script rather than by a person at a terminal.

before-image journal

A file that contains copies of records before they have been updated. VAX DBMS and Rdb/VMS use before-image journaling to automatically undo updates to a database when a transaction is rolled back. Before-image journaling is also called recovery-unit journaling or short-term journaling.

block step

One of three kinds of steps used to define the work of a multiple-step ACMS task. A block step has three parts: attributes, work, and action.

Boolean expression

A string that specifies a condition that is either true or false. For example:

```
PRINT PERSONNEL WITH STATUS = 'TRAINEE' AND AGE LT 30
```

Here, the Boolean expression is STATUS = 'TRAINEE' AND AGE LT 30. The PRINT statement displays only those records for which the value of this expression is true.

See also Boolean operator, conditional expression.

Boolean operator

A symbol or word that lets you join two or more Boolean expressions. Boolean operators are AND, OR, and NOT. For example, the expression STATUS = 'TRAINEE' AND SALARY > 10000 contains the Boolean operator AND.

broadcast message

A text line that lets you know of a system event, such as a system shutdown or the receipt of mail.

build operation

In TDMS, the execution of the BUILD LIBRARY command in the Request Definition Utility (RDU). This operation places the requests named in the request library definition and their associated form and record information in a request library file. The program accesses this file at run time to execute a request. RDU creates this library file in your default directory with an .RLB file type.

In ACMS, the execution of the BUILD command in the Application Definition Utility (ADU). After you create a definition, you use the ADU BUILD command to create a binary form of the definition for use at run time.

CALC mode

In VAX DBMS, a way to calculate a record's storage address in the database by hashing the value of one or more data items in the record. CALC mode is declared in the storage schema and can be used only with SYSTEM-owned sets.

call interface

A mechanism for a program to access components of a software product. For example, the VAX DATATRIEVE Call Interface is the part of VAX DATATRIEVE that provides access to DATATRIEVE's data management services. There are three modes of access to DATATRIEVE's call interface:

- Through the terminal server
- Through the remote server
- From a calling program

callable DBQ

In VAX DBMS, a data manipulation interface to the Database Control System that lets programs written in any VAX language that conforms to the VAX Calling Standard access a database.

See also Database Query, interactive DBQ.

Callable RDO

A single external routine that accepts an Rdb/VMS statement as a parameter. You can call this routine from any language that adheres to the VAX Calling Standard. Callable RDO lets a program use Rdb/VMS even if no precompiler exists for the language.

calling program

A program that issues calls to other programs or subprograms to execute certain operations.

In VAX DATATRIEVE, for example, a high-level language program that contains calls to callable DATATRIEVE routines is referred to as the calling program.

cancel action

A procedure or image called by an ACMS task when the task is canceled. The cancel action does cleanup work for the task, such as recovering from incomplete operations; it does not release locks or perform other work specific to a server.

cancel procedure

A procedure called by an ACMS task when the task is canceled if, at the time of the cancellation, the task is processing in or keeping server context in a procedure server process. The cancel procedure does cleanup work for the server process, such as releasing record locks, so that the server process can be reused without being restarted. When a cancel procedure is called, it runs in the server process allocated to the task, whether or not the task is using the server process at the time of the cancellation.

candidate key

A field or set of fields that uniquely identifies the individual records of a relation. For example, in a relation of employee information, the employee identification number is a candidate key.

case value

A literal value in a TDMS request that determines whether a conditional instruction executes at run time. TDMS checks that the case value matches the value in a control field. If there is a match, TDMS executes the request instructions associated with the case value.

CDD

See Common Data Dictionary.

CDDL

See Data Definition Language Utility.

CDDL source file

A file in which you define CDD records. The CDDL compiler inserts the definitions in a CDDL source file into the CDD directory hierarchy.

CDDV

See Dictionary Verify/Fix Utility.

CHAIN mode

In VAX DBMS, a way to link records sequentially using NEXT, PRIOR, and OWNER pointers. CHAIN mode is declared in the storage schema and cannot be used with CALC sets.

character string

A string of characters (bytes) that is identified by an address and a length.

child

A way of describing a dictionary directory, subdictionary, or object in the CDD that immediately follows another directory or subdictionary in the CDD hierarchy. For example, given CDD$TOP and CDD$TOP.MANUFACTURING, CDD$TOP is the parent and CDD$TOP.MANUFACTURING is the child.

See also ancestor, descendant, and parent.

cluster

See VAXcluster.

CLUSTERED VIA set option

In VAX DBMS, a record placement option in which the Database Control System (DBCS) stores records on or near the page that contains the owner of the set. The CLUSTERED VIA option is declared in the storage schema definition.

CODASYL

An acronym for the Conference on Data Systems Languages, the committee that designed the COBOL language and provided the guidelines used in the development of DBMS. VAX DBMS is CODASYL compliant.

CODASYL-compliant

Any database system that conforms to the guidelines set by the Conference on Data Systems Languages.

collating sequence

The sequence in which characters are ordered for sorting, merging, and comparing.

collection

- In VAX DATATRIEVE, a type of record stream formed with the FIND statement. You can name a collection in order to have several collections available at once.
- In VAX DBMS, all occurrences of records that belong to a specific record type. Record types are defined in schema and subschema entries.

See also CURRENT.

column

A relational database term used interchangeably with field.

See also field.

column headers

The heading that labels the columns of data in a DATATRIEVE report or in the output of a DATATRIEVE PRINT statement.

command process (CP)

The process in the ACMS terminal control subsystem that handles user logins and the interaction between terminals and ACMS.

See agent.

comment character

A character that begins a line of descriptive text in a program or procedure; it does not affect program or procedure execution. Comment fields begin with a comment character reserved by the language you are using. Some typical comment characters are the exclamation point (!), the asterisk (*), and the letter C. Comment fields end with a carriage return.

COMMIT

- In VAX DBMS, a statement that ends a transaction, making permanent all database updates temporarily stored by the recovery unit journal file.
- In Rdb/VMS, a statement that ends a transaction, entering all database changes in the physical database file.

- In ACMS, the function that ends a recovery unit and makes permanent database operations performed in the recovery unit. Also an Application Definition Utility keyword used in defining ACMS tasks with database recovery.
- In DATATRIEVE, a statement that makes permanent all changes to Rdb/VMS and VAX DBMS databases since the most recent COMMIT or ROLLBACK statement or since the first READY command if you have not issued a COMMIT or ROLLBACK statement.

See also RETAINING and ROLLBACK.

Common Data Dictionary (CDD)

A central storage facility consisting of a hierarchy of directories that contain definitions used by VAX Information Architecture components. The CDD contains descriptions of data, not the data itself. CDD objects are stored hierarchically and are accessed by reference to dictionary path names.

CDD directories and subdictionaries contain directories and objects such as:

- ACMS application, menu, task, and task group definitions
- DATATRIEVE domain, record, plot, procedure and table definitions
- VAX DBMS schema, subschema, storage schema, and security schema definitions
- TDMS record and form definitions, requests, and request library definitions
- Rdb/VMS database, relation, field, index, and constraint definitions

COMPUTED BY fields

Virtual fields that appear in a DATATRIEVE record definition or an Rdb/VMS relation or view definition, but not in the physical record. Because the value of a COMPUTED BY field is computed as part of a statement, it occupies no space in the record.

concurrency

The simultaneous use of a database by more than one user.

conditional expression

A string that can be evaluated as true or false. For example, in the statement FOR E IN EMPLOYEES WITH E.STATE = "MA", the conditional expression is E.STATE = "MA". A conditional expression can also use the Boolean operators AND, OR, and NOT.

See also Boolean expression.

conditional instruction

A TDMS request instruction that executes only if certain conditions are true. TDMS executes a conditional instruction if the value in a control field matches the case value specified within the conditional instruction.

conditional request

A request containing one or more conditional instructions.

constraint

A set of criteria that restricts the values in a field. In Rdb/VMS, you set up constraints using the DEFINE CONSTRAINT statement.

context variable

A temporary name that identifies a record stream to Rdb/VMS and DATATRIEVE.

Once you have associated a context variable with a relation or a domain, you use only that context variable to refer to records from that relation in the record stream or loop you created. In this DATATRIEVE statement, D is a context variable:

```
FOR D IN PERSONNEL PRINT D.DEPT
```

You can also use context variables in DATATRIEVE to resolve context ambiguity.

control field

A program record field specified in a TDMS request whose value determines whether or not TDMS executes a conditional instruction.

See also conditional instruction.

cross operation

See join operation.

cross product

A relation that is the result of performing a join operation to combine every row in one relation with every row in another.

currency indicators

VAX DBMS pointers that serve as place markers in the database for the Database Control System (DBCS) and your run unit.

CURRENT

- In VAX DBMS, identifies the most recently retrieved or updated database records, sets, and realms.
- In DATATRIEVE, identifies the most recently formed collection.

Data Definition Language Utility (CDDL)

The VAX CDD utility that lets you insert record definitions into the CDD. You create the data descriptions in a CDDL source file, and you compile the source file with the CDDL compiler.

data definition language (DDL)

In VAX DBMS, a language used to describe schemas, subschemas, storage schemas, and security schemas.

In VAX Rdb/VMS, a set of statements that let you define the structure and characteristics of stored data. You use the data definition language to describe fields, relations, views, indexes, and constraints. The Rdb/VMS data definition language is part of RDO, the interactive Rdb/VMS utility.

data item

In Rdb/VMS, the smallest unit of data. A data item occupies a single field in a record.

data item occurrence

In VAX DBMS, one occurrence of a data item type.

See also data item type and record occurrence.

data item type

In VAX DBMS, the smallest unit of named data in a record type. A data item can be a single value or an array of one or more values.

See also data item occurrence and record type.

Data Manipulation Facility (DMF)

The part of DATATRIEVE that parses, optimizes, and executes all commands and statements passed to DATATRIEVE.

data manipulation language (DML)

In VAX DBMS, the statements that let programs written in VAX languages access the database.

In Rdb/VMS, a set of statements that lets you store, retrieve, modify, and erase data from a database. Rdb/VMS provides two methods of manipulating data:

- Embed the data manipulation statements in a high-level language, such as COBOL.
- Issue the data manipulation statements directly, using the RDO utility.

data type

A characteristic assigned to a field that determines the kind of data the field can contain.

data value

A user-assigned value of a data item occurrence.

See also data item occurrence and data item type.

database

A collection of interrelated data on one or more mass storage devices. The collection is organized to facilitate efficient and accurate inquiry and update.

In a database, more than one user can access the data at the same time. Data integrity and security are provided by the database.

See also hierarchical database, network database, and relational database.

Database Control System (DBCS)

The VAX DBMS or Rdb/VMS component that, together with the VMS operating system, provides run-time control of database processing.

database handle

A name given to an Rdb/VMS database to distinguish it from other currently active databases in a program.

database key (dbkey)

In VAX DBMS and Rdb/VMS, a unique value that identifies the address of a record in a database. The Database Control System assigns the value when a record is stored in the database.

In VAX DBMS, database keys are used by the Database Control System whenever you store, retrieve, or manipulate a record.

In Rdb/VMS, your program can retrieve the database key and use it to access a record.

database management system

A system for creating, maintaining, and accessing a collection of interrelated data records that may be processed by one or more applications without regard to physical storage. Data is described independently of application programs, providing ease in application development, data security, and data visibility.

The VAX Information Architecture includes two database management systems:

- VAX DBMS, a database management system that complies with the standards established by CODASYL
- Rdb/VMS, a database management system based on the relational data model

See also database and relational database.

database pages

The structures used to store and locate data in a VAX DBMS or Rdb/VMS database. Database pages consist of one or more disk blocks of 512 bytes each.

VAX DBMS uses page-clustered I/O, a technique that retrieves groups of physically-related database pages, rather than an individual page, in response to a run unit's request for data.

Database Query (DBQ) Utility

In VAX DBMS, a data manipulation utility that interprets data manipulation language (DML) statements. DBQ provides access to data through both interactive and callable modes. Interactive DBQ lets you test the logic of DML statements before including the statements in a program. Callable DBQ provides access to the database for programs written in high-level languages.

See also callable DBQ and interactive DBQ.

DATATRIEVE

A VAX query language and data management tool for manipulating, storing, and modifying records from RMS data files, VAX DBMS databases, and Rdb/VMS databases. DATATRIEVE also generates reports and graphs from data stored in RMS files, VAX DBMS databases, and Rdb/VMS databases. DATATRIEVE is callable from a variety of high-level languages.

DATATRIEVE procedure

See procedure.

DBCS

See Database Control System.

dbkey

See database key.

DBMS

Used generically, DBMS can refer to any database management system. In VAX Information Architecture documentation, DBMS usually refers to VAX DBMS, a DIGITAL software product that complies with the standards for database management systems established by CODASYL.

See also CODASYL, database management system.

DBQ

See Database Query Utility.

DBR

In VAX DBMS, the name of the process that performs database recovery. It is called by the DBMS monitor.

DCL

DIGITAL Command Language.

DCL command procedure

A sequence of DIGITAL Command Language (DCL) commands stored in a file; sometimes referred to as a DCL procedure.

DCL server

One of two types of servers used to handle processing work for ACMS tasks. A DCL server handles images, DATATRIEVE commands, and DCL commands and command procedures.

See also server, DCL server image, and procedure server.

DCL server image

The image, provided by ACMS, that is loaded into a DCL server process when the process is started by the application execution controller. The DCL server image lets you use images, DATATRIEVE commands, and DCL commands and procedures to implement processing for tasks.

See also procedure server image.

DCL server process

See server process.

DDL

See data definition language.

DDU

See Device Definition Utility.

deadlock

A situation in which two or more processes request the same set of resources and there is no method for resolving the conflict. For example, if process A has record 1 locked and requests record 2 while process B has record 2 locked and is requesting record 1, a deadlock occurs between processes A and B.

DECnet

The DIGITAL software facility that lets a user access information on a remote computer through telecommunications lines. DECnet/VAX enables a VMS operating system to function as a network node.

default

A value that is assumed unless or until you specifically indicate another choice.

default dictionary directory

The CDD directory assigned to you when you invoke an image that uses the CDD. This directory becomes the starting directory for path names. You can define a directory as the default by assigning a path name to the logical name CDD$DEFAULT. If you do not, the default directory is CDD$TOP. The CDD Dictionary Management Utility and some command qualifiers let you set temporary default directories. You can also set the default directory with the DATATRIEVE SET DICTIONARY command.

default directory

The directory from which the VMS system reads and to which it writes all files that you create unless you explicitly name a directory.

degree (of a relation)

The number of fields in a relation definition.

descendant

A dictionary directory, subdictionary, or object in the CDD below another directory or subdictionary in the CDD hierarchy.

See also ancestor, child, and parent.

descending order

A method of sorting that starts with the highest value of a key and proceeds to the lowest value, in accordance with the rules for comparing data items.

See also ascending order.

detail lines

The formatted data lines in a DATATRIEVE report or PRINT statement.

Device Definition Utility (DDU)

The ACMS tool for defining which terminals have access to ACMS.

Device Utility

See Device Definition Utility.

dictionary

In the most general sense: an overall hierarchical storage facility that includes dictionary directories, subdictionaries, and objects. In the VAX Information Architecture documentation, dictionary refers to the VAX Common Data Dictionary (CDD).

As a keyword used with the DATATRIEVE or RDO SET and SHOW commands, "dictionary" has the more limited meaning of the current location within the CDD.

See also Common Data Dictionary.

dictionary directory

The structure for organizing data descriptions stored in the CDD. Dictionary directories are similar in function to VMS directories. They "own" other dictionary directories or dictionary objects.

See also default dictionary directory.

Dictionary Management Utility (DMU)

The Common Data Dictionary (CDD) management utility that lets you create and maintain the CDD directory hierarchy and its associated access control and history lists.

dictionary object

A data definition stored in the Common Data Dictionary (CDD). Examples of objects include:

- VAX DATATRIEVE domains, records, procedures, plots, and tables
- VAX DBMS schemas, areas, sets, and records
- VAX CDD record definitions
- VAX TDMS forms, requests, and request library definitions
- VAX Rdb/VMS relation, view, and field definitions

See also Common Data Dictionary.

Dictionary Verify/Fix Utility (CDDV)

A Common Data Dictionary (CDD) utility that detects damaged dictionary files and repairs them. CDDV also compresses the data in dictionary files for more efficient use of storage.

directory

A file that briefly catalogs a set of files stored on disk or tape. The directory includes the name, type, and version number of each file in the set.

See also default directory.

directory hierarchy

The structure of CDD directories. The hierarchy of dictionary directories, subdictionaries, and objects is a tree structure. Each dictionary directory in the CDD tree may become a parent by owning other dictionary directories or dictionary objects. Dictionary objects are the terminal points of the hierarchy; they cannot be parents.

See also ancestor, child, descendant, and parent.

distributed transaction processing

The processing of ACMS tasks or applications on remote nodes. An ACMS user or task submitter on one node can select tasks or applications from an ACMS system on another node.

DML

See Data Manipulation Language.

DMU

See Dictionary Management Utility.

domain

A DATATRIEVE data structure that associates a name with the relationship between a file and a record definition. Using the domain name gives access to information in the data file as interpreted by the record definition. For example, the domain PERSONNEL associates the file PERSON.DAT and the record definition PERSONNEL_REC.

DYNAMIC allocation

In VAX DBMS, an option that tells the Database Control System to perform data compression on an item occurrence in the database. DYNAMIC allocation is declared in the storage schema.

edit string

A character or group of characters that controls how DATATRIEVE displays data in a field. Edit strings can differ from record definition picture clauses. Picture clauses control how DATATRIEVE stores data in a field.

elementary field

A record segment containing one item of information. It might contain a department number, a last name, or any other information you want to define as a single item.

See also field.

exchange step

One of three kinds of steps that define the work of a VAX ACMS multiple-step task. An exchange step handles input and output between the task and the task submitter.

See also processing step and block step.

execution controller

See application execution controller.

explicit mapping

The TDMS request instructions (COPY TO, INPUT TO, OUTPUT TO, and RETURN TO) that you use to map data between form and record fields.

See also implicit mapping.

external file

A file opened in a main procedure that is accessed from an external subroutine.

external subroutine

A procedure that can be compiled separately from a main procedure.

field

A single division within a record where a data item is stored. You define a field's name and data type, along with other characteristics, using a data definition language.

See also elementary field and group field.

field attribute

A condition or characteristic of a field in a record.

See also form field attribute.

field constant

See form field constant.

field description statement

The statement that defines field characteristics in CDDL source files. The four types of field description statements are ELEMENTARY, STRUCTURE, COPY, and VARIANT.

field identifiers

See picture characters.

field name

The name given to a particular field in a record or form.

field picture

See picture string.

file

A collection of related records.

file name

The name you choose to identify a file. The file name can have from 1 to 39 characters selected from the letters A through Z, the numbers 0 through 9, and an underscore (_) or a dollar sign ($). When you name files, you can use any names that are meaningful to you.

file specification

A name that uniquely identifies a file. A full file specification identifies the node, device, directory name, file name, type, and version number under which a file is stored.

file type

The part of a file specification that describes the nature or class of file. The file type follows a period after the file name and consists of from 1 to 39 characters. VMS recognizes many default file types used for special purposes. For example, .RDB is the default file type for an Rdb/VMS database.

FIXED member

In VAX DBMS, a record occurrence that, upon becoming a member of a set occurrence of a set type, must remain a member of that set until the record occurrence is erased from the database. Fixed set membership is specified in a schema DDL entry.

See also MANDATORY member and OPTIONAL member.

foreign key

In Rdb/VMS, a key that does not uniquely identify records in its own relation but is used as a link to matching fields in other relations. For example, DEPARTMENT_CODE is included in the JOB_HISTORY relation in order to link it to the DEPARTMENTS relation.

form

A terminal screen image used to display and collect information.

See also form definition.

form attributes

Characteristics assigned to an entire form. Examples of form attributes are screen background and screen width.

form definition

A description of a form, created with the TDMS Form Definition Utility (FDU) and stored in the CDD. A form definition can contain information that identifies:

- The screen image of the form, including the location of background text, fields, and video highlighting
- The length or size, data type, and edit type of each field
- A set of attributes for each field on the form

Form Definition Utility (FDU)

The TDMS utility used to process (create, modify, replace, or copy) form definitions and to store them in the CDD. You also use this utility to provide a listing of form definitions used in an application.

form field attribute

In TDMS, a condition or characteristic applied to a form field during form definition. Requests can override some TDMS form field attributes. Examples of form field attributes are video characteristics and requirements for filling a field.

form field constant

A character or embedded space that is displayed in a field on a TDMS form at run time. For example, you can use a hyphen as a field constant in a field that represents a telephone number.

free space

In VAX DBMS, the sections of the database page that are not used.

See also database pages.

full path name

A name that uniquely identifies a dictionary directory, subdictionary, or object in the CDD hierarchy. The full path name is a concatenation of the given names of directories and objects, beginning with CDD$TOP, ending with the given name of the object or directory you want to specify, and including the given names of the intermediate subdictionaries and directories. The names of the directories and objects are separated by periods.

CDD$TOP.DEPT32.EMPLOYEE is a full path name that uniquely identifies the object EMPLOYEES.

See also path name, relative path name, and given name.

given name

The name assigned to a dictionary directory, subdictionary, or object in the Common Data Dictionary (CDD). A given name contains up to 31 characters from the set A-Z, 0-9, _, and $. The first character must be a letter from A-Z, and the last character cannot be _ or $. If you are using a VT200-family terminal, you can use 8-bit alphabetic characters.

The root directory, having only descendants but no ancestor, has the given name CDD$TOP. The given names of all other directories and objects are assigned by the user creating them. The given name of a dictionary directory, subdictionary, or object is separated from the name of its parent by a period.

In the path name CDD$TOP.DEPT32.EMPLOYEE, EMPLOYEE is the given name of a CDD object and DEPT32 is the given name of a dictionary directory.

global aggregate

In Rdb/VMS, an expression that uses field values from one relation to group records from another. A statistical expression is then used to calculate a value for the group. For example, you can group salary records in a SALARY_HISTORY relation according to the DEPARTMENT_CODE field in the DEPARTMENT relation. Then you can use the AVERAGE function to find the average salary for each department.

GOLD key

The PF1 key on the numeric keypad that you use in combination with some of the other keypad and arrow keys.

group data item

In VAX DBMS, a named entity that contains one or more data item types. Group data items are declared in a subschema definition.

group field

A field in a record containing other fields. A group field can contain one or more elementary fields or other group fields.

A group field in DATATRIEVE is equivalent to a group data item in DBMS and a STRUCTURE field description in CDDL.

See also field, group data item, and STRUCTURE field description statement.

group record array

In TDMS, a record array whose elements contain other fields. Each of these fields has the same characteristics (length, data type, and so on), and each field is referred to in a request by the same name, but with a unique subscript. In request instructions, you can include the group array field name to make each field name unique.

group workspace

A workspace that holds information needed by many tasks in an ACMS task group. A group workspace is made available when the first task in a group that uses that workspace is selected by a terminal user. Once allocated, a group workspace remains available to all tasks in the group until the application stops.

See also task workspace, user workspace.

hashing

In VAX DBMS, the conversion of a data item value (for example, a key value) into a fixed-length numeric value using a special algorithm. Hashed key values are used as pointers to database record occurrences.

hierarchical database

A type of database that organizes the relationships between record types as a tree structure. A hierarchical database stores related records on the same branch of the tree to make data retrieval efficient.

See also database, network database, relational database.

history list

An optional audit trail maintained by the CDD to monitor the processing and use of dictionary directories, subdictionaries, and objects.

See also audit trail.

image

A file consisting of procedures and data that have been bound together by the linker. There are three types of VMS images: executable, shareable, and system. When not otherwise stated, image refers to an executable image.

implicit mapping

Using the TDMS mapping instructions INPUT, OUTPUT, and RETURN with the %ALL parameter to map data between all identically named form and record fields. You can include implicit mapping instructions in requests to map data without naming individual fields. If the Request Definition Utility finds an error when mapping with the %ALL parameter, it does not include that mapping when it stores the request in the CDD. At run time, TDMS performs only the correct mappings.

See also explicit mapping.

index

A structure within a file or database that lets you locate particular records based on key field values.

index key

A field of a record in an indexed file or database that determines the order of search and retrieval.

- An RMS indexed file has one primary key and optionally one or more alternative keys.
- In DATATRIEVE, you declare an index key for RMS files in the DEFINE FILE command, by naming a field from a record definition.
- In Rdb/VMS, you can use any field or combination of fields from a record as an index key. You can also define more than one key for a given relation.
- In VAX DBMS, you can use any field or combination of fields from a record as an index key for a sorted set.

INDEX mode set

In VAX DBMS, a sorted set in which a hierachical index data structure is used to speed access to a specified record occurrence. INDEX mode is specified in a storage schema DDL entry.

See also index node.

index node

In VAX DBMS, the index data structure for an INDEX mode set.

See also INDEX mode set.

indexed file

An RMS file that has a primary key and optionally one or more alternative keys.

indexed form array

See group record array.

initialization procedure

In ACMS, a procedure that runs when a procedure server process starts and that opens files or readies a database for the server process.

INSERTION class

In VAX DBMS, an attribute of member record types that describes how and when member record occurrences are added to set occurrences.

See also AUTOMATIC member and MANUAL member.

Installation Verification Procedure (IVP)

A command file that tests whether a software product has been installed correctly.

integrity

The correctness of information in an Rdb/VMS or VAX DBMS database. There are three general types of integrity control:

- Integrity constraints make sure that database information remains correct when users try to modify it incorrectly.
- Concurrency control lets only one user at a time update a file while allowing many users simultaneous access to the database.
- Recovery restores a database to a the state it was in before a system failure.

interactive DBQ

In VAX DBMS, a data manipulation interface to the Database Control System that allows low-volume, interactive access to a database. You can use interactive DBQ as a tool to test and debug program logic. When used on a VT100- or VT200-series terminal, interactive DBQ uses a split screen to show your current position in a subschema after each DML statement is executed.

See also callable DBQ, Database Query Utility.

interactive processing

A mode of computer operation in which the commands and data that control the actions of the computer are entered by a person at a terminal.

interpretive call interface

See Callable RDO.

join operation

A procedure that selects a record from one relation, associates it with a record from another relation, and presents them as though they were part of a single record.

journal file

In VAX DBMS and Rdb/VMS, a file that contains all records modified by a run unit or transaction. The journal file allows reconstruction of the database in the event of corruption due to system or program failures.

journaling

The process of recording on a recoverable resource information about operations on a database. The type of information recorded depends on the type of journal being created.

See also after-image journal and before-image journal.

junction record

In VAX DBMS, a record that relates two records to each other. You can use a junction record to define a recursive or many-to-many relationship between two records.

keeplist

In VAX DBMS, a list of database keys used to recall their associated records. Database keys are placed on and removed from keeplists at the direction of a DML operation.

key

In Rdb/VMS, a field in a record that you use to define an index. Using index keys, Rdb/VMS can locate records in the relation directly, without searching sequentially. Defining index keys increases the speed of some database operations.

In VAX DBMS, a field or combination of fields in a record that defines a sort key for an INDEX mode set or a hashing key for a CALC mode set.

See also candidate key, foreign key, index key, key value, and primary key.

key value

In VAX DBMS, the values supplied in a DML operation to identify a specific record for access.

keyword

A word reserved for use in certain specified syntax formats, usually in a command or a statement.

line graph

Line graphs and scattergraphs compare values in fields and expressions by plotting values as points on X and Y axes. Line graphs connect the points; scattergraphs plot only the points.

See also bar chart and pie chart.

line index

In VAX DBMS, a dynamic section of a database page that acts as a directory to data on the page.

See also database pages.

linker

A program that creates an executable program, called an image, from one or more object modules produced by a language compiler or assembler. Programs must be linked before they can be executed.

literal

A value expression representing a constant. A literal is either a character string enclosed in quotation marks, or a number.

load file

In VAX DBMS, an RMS file containing data and set-significant information used by the database load facility.

locking

In VAX DBMS and Rdb/VMS, the facility that controls the allocation and deallocation of a resource, such as a record or a process. VAX DBMS allows locks on individual records and entire realms.

logical operator

See Boolean operator.

logical path name

A logical name that uniquely identifies a dictionary directory, subdictionary, or object in the Common Data Dictionary (CDD) hierarchy. The logical path name is a name you define for a full or relative path name. For example you may define a logical path name using the DCL DEFINE command:

```
$ DEFINE EMP CDD$TOP.DEPT32.EMPLOYEE
```

Then, within the current process, EMP is equivalent to the full path name CDD$TOP.DEPT32.EMPLOYEE.

MANDATORY member

In VAX DBMS, a record occurrence that, upon becoming a member of a set occurrence of a particular set type, must remain a member of that or some other occurence of that set type until the record is erased from the database. MANDATORY set membership is specified in a schema entry. A MANDATORY member can be moved from one set occurrence to another within the same set type.

See also FIXED member and OPTIONAL member.

MANUAL member

In VAX DBMS, a record occurrence that becomes a member of a specific set occurrence by direction of an application program. MANUAL set membership is specified in a schema entry.

See also AUTOMATIC member.

mapping

The description of the exchange of data between a TDMS form and a program record.

See also explicit mapping, implicit mapping.

MDB

See menu database.

member record

In VAX DBMS, a record, other than an owner record, included in a set. There may be one or more member record types in a set, and zero or more member record occurrences. A member record must be accessed through its owner record occurrence or the SYSTEM record.

See also owner record and nonsingular set type.

menu

A list of tasks, from which a user selects one for processing. A menu can also direct users to other menus.

In ACMS, you define the list of items on a menu and other menu characteristics using the Application Definition Utility (ADU).

menu database (MDB)

A run-time database containing information derived from menu definitions. ACMS uses the information in the menu database for displaying menus. A menu database is created by building menu definitions with the Application Definition Utility (ADU).

message file

A file that contains a table of message symbols and their associated text.

metadata

Data that is used to describe other data. Data definitions are sometimes referred to as metadata.

multiple-step task

An ACMS task defined in terms of a block step that contains one or more exchange and processing steps.

See also block step, exchange step, processing step, and step action.

navigation

In VAX DBMS, the process of traversing database records along a hierarchical path.

network database

A database model that establishes relationships between records using sets. A single record can participate in any number of sets, so you can relate it to any other record in the database, not just those above and below it in a hierarchy.

Network databases are also called CODASYL databases.

See also database, hierarchical database, relational database.

nonsingular set type

In VAX DBMS, a set type owned by a user-defined record type, not by the SYSTEM record.

See also member record, owner record, and SYSTEM-owned set type.

normalization

The process that reduces a database structure to its simplest form and eliminates data redundancy. Normalization physically separates related concepts in the database into separate relations or records. A data item is stored only once and requires only one update operation to change it.

Novalidate mode

The mode in the TDMS Request Definition Utility that lets you create and store a request without checking for correct mappings and references. You create a request in Novalidate mode by using the SET NOVALIDATE command. Validate mode is the default.

See also validation.

numeric data type

A characteristic assigned to a field that indicates field values are to be considered numbers rather than text.

object

See dictionary object.

operator command

See ACMS Operator command.

OPTIONAL member

In VAX DBMS, a record occurrence that can be removed from all set occurrences. You can change its set membership without deleting it from the database. OPTIONAL set membership is specified in a schema entry.

See also FIXED member and MANDATORY member.

owner record

In VAX DBMS, the owner records serve as access entry points to set occurrences. Only one record type can be the owner for each set type, and only one owner record occurrence can be the owner of each set occurrence.

See also member record, nonsingular set type, and SYSTEM-owned set type.

page header

In VAX DBMS, a fixed-length section at the beginning of the database page that contains page and storage area information.

See also database pages.

parent

The dictionary directory or subdictionary in the CDD that immediately precedes a directory, subdictionary, or object in the CDD hierarchy. A parent can have many children, but each dictionary directory, subdictionary, and object in the CDD can have only one parent. For example, CDD$TOP is the parent of CDD$TOP.MANUFACTURING. CDD$TOP.MANUFACTURING is owned by CDD$TOP.

See also ancestor, child, and descendant.

partial path name

See relative path name.

path name

A unique designation that identifies a dictionary directory, subdictionary, or object in the CDD hierarchy. The full path name combines the given names of directories and objects, beginning with CDD$TOP, ending with the given name of the object or directory you want to specify, and including the given names of the intermediate subdictionaries and directories. The names of the directories and objects are separated by periods.

You can have full, logical and relative path names.

See also full path name, given name, logical path name, and relative path name.

picture characters

The characters specified in a record or form definition that determine the length and type of a field. For example, a C in the form definition of a TDMS form field indicates that only an alphanumeric character (A-Z, a-z, 0-9, space) can be entered in that field. The group of picture characters that make up a field is called the picture string.

picture clause

A clause in a record definition that describes how data for a field should be stored. For example, the following DATATRIEVE field definition contains a picture clause specifying that values of no more than 20 characters of alphanumeric data can be stored in ADDRESS:

```
10   ADDRESS    PIC X(20).
```

picture string

A group of one or more picture characters in a record or form definition that determines the location, length, and type of a field. In TDMS for example, 99999 is a picture string that indicates that up to five numeric characters can be entered in that form field.

pie chart

Pie charts compare values in fields or expressions by representing quantities as wedge-shaped percentages of a whole pie.

See also bar chart, line graph, and scattergraph.

PLACEMENT mode

In VAX DBMS, a storage method by which the Database Control System determines the database key values associated with record occurrences based on user-specified set options. PLACEMENT mode is declared in the storage schema.

See also SCATTERED set option and CLUSTERED VIA set option.

plot

A graphic representation of data using DATATRIEVE's graphics capability. You can create three basic kinds of plots using DATATRIEVE:

- Bar charts
- Line graphs and scattergraphs
- Pie charts

pointer

In VAX DBMS, a place marker that identifies a record's address in a storage area.

See also database key.

precompiler

A utility that reads data manipulation language statements in a high-level language program and translates those statements into calls to low-level database routines. Rdb/VMS and VAX DBMS have separate utilities that perform this function.

primary key

In an RMS indexed file, the index key whose value determines the order of records. You cannot modify or erase the value in a primary key field of a DATATRIEVE record.

print list

One or more value expressions (including the names of elementary and group fields) whose values you want DATATRIEVE or Rdb/VMS to display. A DATATRIEVE print list can also include optional formatting specifications.

privilege

The ability to access a file or other resource for a certain purpose. Thirteen privileges have been defined to control access to the CDD. Four of these privileges are specific to VAX DATATRIEVE; the remaining nine are VAX CDD access privileges.

See also access control list.

procedure

- A general purpose routine, entered by means of a call instruction, that uses an argument list passed by a calling program and uses only local variables for data storage. A procedure is entered from and returns control to the calling program.
- A fixed sequence of DATATRIEVE commands, statements, clauses, or arguments that you create, name, and store in the Common Data Dictionary.
- A series of Rdb/VMS RDO statements stored in a VMS file. These can be executed with the execute (@) directive.

See also step procedure, initialization procedure, and termination procedure.

procedure server

One of two types of servers that handle processing work for ACMS tasks. Procedure servers do processing work for step procedures called in tasks defined with ACMS.

See also server, DCL server, and procedure server image.

procedure server image

The image that is loaded into a procedure server process when the process is started by the ACMS application execution controller. The procedure server image is created when all the procedures handled by the server are linked together with the procedure server transfer module for that server.

See also DCL server image and procedure server transfer module.

procedure server process

See server process.

procedure server transfer module

The object module created for a procedure server as a result of building an ACMS task group definition. When a task group is built, the Application Definition Utility produces a procedure server transfer module for each server defined in the task group. The procedure server transfer module is linked together with all the procedures handled by the server to produce the procedure server image.

process

The entity scheduled by the VMS system software that provides the context in which an image runs. A process consists of an address space and both hardware and software context.

process context

See server context.

processing step

One of three kinds of steps that define the work of a task defined with ACMS. The work of a processing step is handled by a server and can consist of computations, data modification, and file and database access.

See also block step and exchange step.

program request key (PRK)

In TDMS, a key or combination of keys that let the terminal operator communicate with the application program at run time. You define the program request key in a request.

A PRK can be defined by either:

- The keyword KEYPAD followed by one key (0-9, hyphen, period, or comma)
- The keyword GOLD followed by one printable key from the main keyboard (including the space bar, but not the tab key)

project operation

See reduction operation.

prompting expression

An expression that directs DATATRIEVE to ask the user to supply a value when a statement is executed.

qualifier

A portion of a command string that modifies a command verb or command parameter. A qualifier follows the command verb or parameter to which it applies and has the following format: /qualifier[=option].

query header

A substitute column header that you define to replace the field name when DATATRIEVE displays values from a field. For example, you may want to define the query header "Status" to appear at the top of the column of values from the field EMPLOYEE_STATUS.

query name

A synonym you give to a DATATRIEVE field name in order to make input easier to type and remember. For example, to make it easier to write DATATRIEVE statements about the field SECTION_NUMBER, you can define the query name NUM and substitute it for the full field name.

quiet point

In VAX DBMS, a time when a run unit is not accessing any database areas. Quiet points occur between transactions.

See also transaction.

Rdb/VMS

VAX Rdb/VMS is a DIGITAL relational database management product, layered on VMS, that uses the relational model of database organization.

RDO

See Relational Database Operator.

RDU

See Request Definition Utility.

RDU commands

The commands you issue to operate the TDMS Request Definition Utility, including commands to process (create, modify, copy, delete, and so on) a request or a request library definition.

realm

In VAX DBMS, one or more areas grouped to allow subschema access. Realms are specified in a subschema entry.

See also area.

record

A body of related information that is the basic unit for storing data.

See also field, member record, owner record, record occurrence, and record type.

record definition

The description of a record's structure that includes the name, data type, and length of each field. CDDL, DATATRIEVE, and DBMS all store record definitions in the CDD. ACMS, TDMS, COBOL, BASIC, DIBOL, FORTRAN, PL/I, and RPG II access record definitions stored in the CDD.

record locking

A process by which a database management system reserves a record or set of records for use by one user, at the exclusion of other users. Record locking helps guarantee the consistency of data.

Record Management Services (RMS)

A set of VMS operating system procedures that programs can call to process files and records within files. VAX RMS lets programs issue GET and PUT requests at the record level (record I/O) as well as read and write blocks (block I/O). RMS is an integral part of the VMS system software and is used by high-level languages, such as VAX COBOL and BASIC, to implement their input and output statements.

DATATRIEVE uses VAX RMS to create, define, store, and maintain files and records within files.

record occurrence

In VAX DBMS, an instance of a record type. A record occurrence is the physical representation of a record; a record type is the logical definition of a record.

See also data item occurrence and record type.

record selection expression (RSE)

A phrase that defines specific conditions that individual records must meet before Rdb/VMS, VAX DBMS, or DATATRIEVE includes them in a record stream. The RSE lets you determine the subset of records to be selected from a set of domains or a database.

record stream

A group of records formed by a record selection expression.

In Rdb/VMS, you form a record stream with either a FOR statement or a START_STREAM statement. Streams are used in an application program or RDO to retrieve one record at a time for manipulation.

In DATATRIEVE, you form a record stream by including an RSE in a DATATRIEVE statement.

See also record selection expression (RSE).

record type

In VAX DBMS, the logical definition of a record. Record types are declared in the schema data definition.

See also data item type and record occurrence.

recovery

In VAX DBMS and Rdb/VMS, the process of restoring a database to a known condition after a system or program failure.

In ACMS, you can define recovery as a characteristic for a multiple-step task that uses VAX DBMS.

See also after-image journal, before-image journal, journal file, journaling, and transaction.

recovery-unit journal

See before-image journal

reduction operation

In Rdb/VMS and DATATRIEVE, an operation that finds the unique values for a field or group of fields and eliminates repeated records. Reduction is sometimes called the project operation. You use the REDUCED TO clause to perform the operation.

reflexive join

An operation that joins a relation to itself.

See also join.

relation

A method of presenting related data that consists of a set of rows and columns. The columns have names and divide each row into fields. For a single field in a row, there is only one data item. In VAX Rdb/VMS, columns are referred to as fields, and rows are called records. A relation is sometimes called a table.

relational database

A database model that represents data as a set of independent tables. Within a table, data is organized in columns and rows, with at most one data item occupying each intersection. Relationships between tables depend on values within the relations. In VAX Rdb/VMS, these tables are called relations.

Relational Database Operator (RDO)

A single interactive utility for maintaining the database, creating and modifying definitions of database elements, and storing and manipulating data.

See also Callable RDO.

relational join operation

See join operation.

relational operator

A symbol, keyword, or phrase you can use to compare values. For example, the DATATRIEVE statement FIND PERSONNEL WITH SALARY > 10000 contains the relational operator > (greater than).

relative path name

The shortened form of a dictionary path name. It includes only the parts of the path name that follow the default CDD directory name. You can use either the full path name or the relative path name to refer to directories, subdictionaries, and objects in the CDD.

See also given name and path name.

remote server

The part of ACMS, DATATRIEVE, DBMS and Rdb/VMS that lets you access data on other computers. If, for example, you are using the computer VACKS1 and you type READY PERSONNEL AT VACKS2, DATATRIEVE logs on to an account on VACKS2. The remote server processes your statements at the remote computer VACKS2.

report header

The heading of a DATATRIEVE Report Writer report, consisting of these optional elements: a centered report-name and, at the top-right corner of the report, a date and a page number.

report specification

A series of DATATRIEVE Report Writer statements that create a report and specify its format.

Report Writer

A subsystem of DATATRIEVE that lets you create reports displaying data in an easy-to-read format.

request

A set of TDMS instructions, created in the Request Definition Utility and stored in the CDD, that describes an exchange of data between a program record and a form. A request includes references to one or more form and record definitions

and instructions for mapping data between a form and a program record. The name of a request is passed as a parameter in the TSS$REQUEST call.

ACMS tasks use requests to display forms on a terminal and gather information from a terminal user.

request call

The call in a TDMS application program that executes a request.

Request Definition Utility (RDU)

The TDMS utility used to process (create, modify, replace, and so on) requests and request library definitions and to store them in the CDD. You also use this utility to build request library files, which are accessed by an application program at run time.

request instructions

The statements in a TDMS request that describe the exchange of data between a program record and a form.

These statements can:

- Identify the record definitions and the associated forms for data transfer
- Provide instructions for transferring the data

The request instructions are executed when the TDMS application program issues a TSS$REQUEST call.

request library definition

A definition, stored in the CDD, that lists the names of related requests to use in a particular TDMS application. A request must be named in a request library definition before you can build a request library file. The program uses the request library file to access requests.

request library file

A VMS file that contains TDMS requests and the form and record information necessary to execute those requests. When you use the Request Definition Utility to build a request library file, RDU reads the definitions in the CDD and puts information in the request library file so that the program can execute the requests. A request library file that contains a request named in a TDMS call must be opened before a program can use the request. Request library files take the default file type .RLB.

request library instructions

The statements in a TDMS request library definition that identify the requests used in a TDMS application. These instructions also give the name of the request library file where these requests and their associated form and record definitions are to be stored.

restore

In Rdb/VMS or VAX DBMS, an operation that rebuilds a database from a saved copy after a hardware or software failure.

restrict

See select operation.

restriction clause

A phrase in the DATATRIEVE record selection expression that lets you specify the maximum number of records making up a record stream.

RETAINING

In VAX DBMS, an option on the DML COMMIT statement. The COMMIT RETAINING statement:

- Does not empty keeplists
- Retains all currency indicators
- Does not release realm locks
- Releases all record locks

RETENTION class

In VAX DBMS, an attribute of member record types that describes when and how a member record occurrence can be removed from a set.

See also FIXED member, MANDATORY member, and OPTIONAL member.

RLB

See request library file.

RMS

See Record Management Services.

ROLLBACK

- In VAX DBMS or Rdb/VMS, the statement that restores a database to an earlier known state using a before-image journal. The rollback process negates updates to the database made by the transaction or recovery unit being rolled back.
- In ACMS, an Application Definition Utility keyword used when defining multiple-step tasks with database recovery.

rollforward

In VAX DBMS and Rdb/VMS, the process of using an after-image journal to restore a database to a known state. This process replaces updates to the database that were lost because a system or program failure required the installation of backup media.

See also recovery.

root dictionary directory

The directory at the top of the VAX CDD hierarchy. The root directory is named CDD$TOP. Every dictionary directory, subdictionary, and object in the CDD is a descendant of CDD$TOP.

row (of a table)

See record, relation.

RSE

See record selection expression.

run unit

In VAX DBMS, an execution of a single program that accesses a database.

SCATTERED set option

In VAX DBMS, a record placement option in which records are evenly distributed throughout database pages, based upon data values in the record. SCATTERED mode is specified in a storage schema entry.

scattergraph

Scattergraphs and line graphs compare values in fields and expressions by plotting values as points on X and Y coordinates. Scattergraphs plot only the points; line graphs connect the points.

See also bar chart and pie chart.

schema

In VAX DBMS, the logical description of a database, including data definitions and data relationships. The schema is written using the schema data definition language (schema DDL).

scientific notation

A way of expressing very large or very small numbers as a constant multiplied by the appropriate power of 10. For example:

.000000009	.9E-8 (9 times 10 to the power of -8)
9000000.	.9E 7 (9 times 10 to the power of 7)

scrolled form array

A list of elements in a scrolled region on a TDMS form, all of which have the same name and the same length and data type. The number of elements in the scrolled region is undefined, and the request can map up to 32,767 elements of data.

scrolled region

An area, specified in the TDMS form definition, that permits the operator to move through many lines on a field and view or enter data, although only a few lines appear at one time on the screen.

security

The protection of the information stored in a database against unauthorized reading, writing, or deletion.

security schema

In VAX DBMS, a definition that describes the data you want to secure.

select operation

In DATATRIEVE and Rdb/VMS, an operation that chooses from domains or relations those records that satisfy a conditional expression. For example, if you want to display employees with salaries greater than $20,000, a selection operation prevents employees records with salaries less than or equal to $20,000 from appearing in the output.

In DATATRIEVE, select operation more commonly applies to using the SELECT statement to select a target record in a collection.

See also record selection expression.

selected record

The record chosen for display or modification by the DATATRIEVE SELECT statement.

sequential file

A RMS file whose records appear in the order in which they were originally written. A sequential file does not have an index. In DATATRIEVE, you cannot delete records from a sequential file.

server

In ACMS, the component that handles processing work for a task. There are two types of servers: DCL servers and procedure servers. The implementation characteristics for a server are defined in a task group definition. The operational characteristics for a server are defined in an application definition.

See also DCL server and procedure server.

server command

The string passed by an ACMS application execution controller to a server process at the start of a processing step. The string identifies what work the server is to perform.

server context

In ACMS, information local to a server process, such as record locks and file pointers. Server process context can be retained from one step to another in a block step but cannot be passed between servers or tasks.

server image

A VMS image that the ACMS run-time system loads into a server process. There are two types of server images: DCL server images and procedure server images.

server process

A VMS process created according to the characteristics defined for a server in an ACMS application and task group definition. Server processes are started and stopped as needed by ACMS application execution controllers.

set

A defined relationship among records in a VAX DBMS database. A set contains an owner record and one or more member records.

See also set occurrence and set type.

set occurrence

In VAX DBMS, a logical occurrence of a set type. A set occurrence consists of one owner record occurrence and zero or more member record occurrences.

set type

In VAX DBMS, a logical definition of a relationship among record types in a database. A set type contains an owner record type, and one or more member record types.

simple record array

See array.

single-step task

An ACMS task that has only a single processing step. Single-step tasks can be defined in a task group or a separate task definition.

See also multiple-step task.

singular set

See SYSTEM-owned set type.

size validators

A field validator on a TDMS form definition that determines the field data type and sets a predefined range for numeric fields. At run time, size validators prevent the operator from entering data that is not within that range.

software event logger (SWL)

The process that records ACMS and TDMS software events that occur during the running of an application program. In order to see the events logged by the SWL, you must use the Software Event Logger Utility Program.

Software Event Logger Utility Program (SWLUP)

The ACMS utility you use to list selected ACMS or TDMS events that were logged by the software event logger.

sort key

A field that forms the basis for sorting. For example, you can rearrange the records in DATATRIEVE's sample domain PERSONNEL according to seniority by typing PRINT PERSONNEL SORTED BY START_DATE.

sorted set

See INDEX mode set.

STATIC allocation

In VAX DBMS, the default allocation option of the ITEM statement of the storage schema entry. Use it to specify the amount of physical storage you want to dedicate to a particular data item type. You make the specification during the definition of the database, but the actual allocation does not occur until the creation of the database.

See also DYNAMIC allocation and storage schema.

statistical expression

- In Rdb/VMS, an expression that takes values from multiple rows of a relation and combines them into a single result. Statistical expressions include AVERAGE, MAX, MIN, COUNT, and TOTAL.
- In DATATRIEVE, statistical expressions let you summarize and calculate statistical values from fields in records. DATATRIEVE statistical expressions include AVERAGE, MAX, MIN, COUNT, RUNNING COUNT, TOTAL, RUNNING TOTAL, and STD_DEV (standard deviation).

step

A part of an ACMS task definition that identifies one or more operations to be performed. Task definitions can have three kinds of steps: block steps, processing steps, and exchange steps. Each step contains clauses that describe the work to be done in that step and the action that follows the work.

See also block step, exchange step, processing step, step action, step work, single-step task, and multiple-step task.

step action

The part of a step definition that tells ACMS what to do after completing the work for that step. These instructions can consist of a single unconditional action or a series of conditional actions based on the value of a field in a workspace.

step label

A name assigned to a step in a multiple-step ACMS task.

step procedure

A type of procedure called in a processing step of an ACMS task. Step procedures handle computations, data modification, and file and database access for processing steps that use procedure servers. Normally, step procedures do not handle input from or output to a terminal.

step work

The part of an ACMS step definition that describes terminal interactions, processing, or both.

storage schema

In VAX DBMS, a description of the physical storage of data in a database. The storage schema is written using the storage schema data definition entry.

stream

In VAX DBMS, an independent access channel between a run unit and a database. Streams let you access multiple subschemas or databases in a single process.

string descriptor

A data structure that specifies the address, length, and data type of a string. String descriptors are passed as arguments to subroutines.

STRUCTURE field description statement

In CDDL, a statement defining fields that are subdivided into one or more subordinate fields. The top-level field description statement for a record is ordinarily a STRUCTURE field description statement.

subdictionary

A dictionary file physically separate from the main dictionary file that functions almost exactly like a dictionary directory. With a subdictionary, you can augment CDD protection with VMS file protection.

subdirectory

A list of files that is grouped one or more levels below the top-level or main VMS directory.

subschema

In VAX DBMS, a user-oriented view of a database. You can tailor a view to meet the needs of a particular programming language or to focus the extent of data a program can access. The subschema can include everything in the corresponding schema or any part of the schema. The subschema is written using the subschema data definition entry.

subscript

A positive integer that indicates the position of an element in a form or record array. For example, in a TDMS request instruction, to refer to the third element of an array LAST_NAME, you use the array field name and the number 3 (indicating the third element): LAST_NAME[3].

substitution directive

An expression in a command or statement passed to DATATRIEVE from a calling program. The substitution directive is replaced by parameters given in the program.

summary lines

Information you can display in a DATATRIEVE report with the AT TOP and AT BOTTOM statements.

SWL

See software event logger.

SWLUP

See Software Event Logger Utility Program.

synchronous call

A call to a TDMS subroutine that performs the entire requested action before your program can continue running. Thus, your program continues only after the completion of the called subroutine.

See also asynchronous call.

system manager

A VMS user responsible for the overall operation of a VMS system. Responsibilities of the system manager include authorizing all users of the system, setting access requirements for all system resources, and running all procedures necessary to ensure the correct and timely operation of the system.

system workspace

A task workspace whose record definition is provided by ACMS. ACMS provides three system workspaces. At run-time, ACMS fills in the contents of the system workspaces for each task selected by a terminal user. These workspaces, like other task workspaces, last only for the duration of a task instance.

See also group workspace, task workspace, user workspace, workspace.

SYSTEM-owned set type

In VAX DBMS, a set owned by a SYSTEM record rather than by a record type you have created. Each SYSTEM-owned record has only one occurrence in the database but can be the owner of many member record types. It allows unassociated record types to be used as entry points to the database.

See also member record and owner record.

table

See relation.

tag variable

An optional variable in CDDL VARIANTS field description statements. The run-time value of the tag variable determines the current VARIANT.

See also VARIANTS field description statement.

task

In ACMS, a unit of work that performs a specific function and that a terminal user can select for processing. Every task belongs to a task group. Some tasks are defined in the task group they belong to; other tasks have separate task definitions. In either case, they are defined with the ACMS Application Definition Utility. The work of a task can be defined as a single processing step or a block step, which consists of a series of exchange and processing steps.

See also single-step task and multiple-step task.

task debugger

An ACMS debugging tool that is primarily for debugging multiple-step tasks that use procedure servers. The task debugger uses task group databases and procedure server images; it does not require application definitions, menu definitions, or a running ACMS system.

task group

One or more ACMS tasks that have similar processing requirements and that are gathered together so they can share resources. A task group definition, created with the Application Definition Utility, defines the servers used by the tasks that belong to the group. It also defines other characteristics and requirements for the tasks in the group, such as workspaces, request libraries, and message files.

task group database (TDB)

In ACMS, a run-time database containing information derived from task and task group definitions. The Task Debugger uses the TDB when debugging tasks; the Application Definition Utility uses the TDB when building an application database. ACMS also uses the TDB when a terminal user selects a task. The TDB is created as a result of building a task group definition with the Application Definition Utility.

task instance

In ACMS, the occurrence of the processing of a task. Each selection of a task is a task instance. Every task instance is given a unique ID by the ACMS run-time system.

task I/O

In ACMS, the communication between a user and a task instance. This communication can consist of VMS terminal I/O or TDMS requests.

task selection string

In ACMS, the string of characters a terminal user types, in addition to the selection keyword or number, when making a selection from a menu.

task submitter

Any authorized ACMS user who selects tasks for processing, provides input for that processing, and receives the results of that processing. Task submitters must also be authorized VMS users.

task workspace

A workspace used mainly to pass information between steps in a multiple-step ACMS task. A task workspace is allocated when a terminal user starts a task and keeps its contents only for the duration of the task instance.

TDB

See task group database.

TDMS

See Terminal Data Management System.

tenant record

Any VAX DBMS record that participates in a set, whether a member or owner.

terminal control subsystem

A set of ACMS-controlled processes that control terminal user access to ACMS. The terminal control subsystem includes two types of processes: the command process or processes and the terminal subsystem controller.

Terminal Data Management System (TDMS)

A VAX product that uses forms to collect and display information on the terminal. TDMS provides data independence by allowing data used in an application to be separated from the application program. ACMS multiple-step tasks use TDMS services to manage terminal input and output.

terminal server

The part of DATATRIEVE that gives you access to DATATRIEVE's interactive data management services.

terminal subsystem controller

The process in the terminal control subsystem that controls which terminals have access to ACMS.

termination procedure

An ACMS procedure that runs when a procedure server process stops and that closes files or releases databases.

Trace facility

The facility that helps you to debug a TDMS application by letting you monitor the action of a TDMS application program at run time.

transaction

An exchange of information between a database user and a database. The operations in a transaction are treated as a group; either all of them are completed at once, or none of them is completed.

In VAX DBMS and Rdb/VMS, a transaction groups a series of statements that perform a task.

- In VAX DBMS, a transaction normally begins with a READY statement and ends with a COMMIT or ROLLBACK statement. However, a transaction may begin with any DML statement, other than READY, if the previous transaction in the run unit ended with a COMMIT statement that contained a RETAINING clause. VAX DBMS transactions include only data manipulation operations.
- In Rdb/VMS, a transaction normally begins with START_TRANSACTION and ends with COMMIT or ROLLBACK. Rdb/VMS transactions can include data manipulation or data definition statements.

See also COMMIT, quiet point, recovery, and ROLLBACK.

tuple

Relational database terminology for a record or row.

type

A characteristic of each element in the CDD. Directories and subdictionaries are directory types, and there are several types of dictionary objects (for example, CDD$RECORD and DTR$DOMAIN).

UDU

See User Definition Utility.

UIC

See user identification code.

unique name

A designation assigned to a component, such as a task, that is used to identify that component within and across definitions.

usage mode

In VAX DBMS, the combination of the DML READY statement's allow mode and the access mode. It describes how a realm or realms you have readied can be used. The eight usage mode combinations are:

BATCH UPDATE	BATCH RETRIEVAL
PROTECTED UPDATE	PROTECTED RETRIEVAL (default)
CONCURRENT UPDATE	CONCURRENT RETRIEVAL
EXCLUSIVE UPDATE	EXCLUSIVE RETRIEVAL

See also access mode and allow mode.

user definition file

A file, created and maintained with the ACMS User Definition Utility, that contains a list of users authorized to access ACMS.

User Definition Utility (UDU)

The ACMS tool for authorizing ACMS users and defining characteristics of those users.

user identification code

A code identifying a user by a group number or name and a member number or name. Both numbers or names are enclosed in brackets.

user name

A designation assigned to a VMS user to identify that user. Also the name a terminal user types to log into VMS and ACMS.

user utility

See User Definition Utility.

user work area (UWA)

In VAX DBMS, a portion of memory assigned to your run unit that holds data to be transferred between your run unit and the Database Control System. It holds data that is either going from your run unit to the database or is coming from the database to your run unit. The UWA also contains definitions of external Database Control System functions.

user workspace

In ACMS, a workspace, defined as an attribute of a task group, that holds information about a terminal user. A user workspace is created the first time a terminal user starts a task that refers to it. ACMS keeps a separate copy of a user workspace for each user and saves the contents of the workspace until the user exits from ACMS.

UWA

See user work area.

valid request

A TDMS request in the Common Data Dictionary (CDD) with the following characteristics:

- The form and record definitions named in the request are stored in the CDD.

- The record field and form field names used in mapping instructions are the same as those contained in the form and record definitions.
- The data types, lengths, and structures of the fields are compatible according to TDMS mapping rules.

validation

The process of checking data on entry to ensure that it meets preestablished requirements.

In DATATRIEVE and Rdb/VMS, for example, the VALID IF clause in the record definition sets criteria for validation of values entered for storage.

When the definition utilities of TDMS and ACMS are in Validate mode, they check that references to external definitions are correct before storing a definition in the CDD.

See also valid request.

value expression

A symbol or string of symbols that you use to calculate a string or numeric value. When you use a value expression in a statement, Rdb/VMS or DATATRIEVE calculates the value associated with the expression and uses that value when executing the statement.

variable

A name associated with an expression whose value can change.

In DATATRIEVE, you use the DECLARE statement to create a variable. For example, the following statement creates a variable, X, that can be assigned any two-digit numeric value: DECLARE X PIC 99.

VARIANTS field description statement

A CDDL statement defining a set of two or more fields that provide alternative descriptions for the same portion of a record. The function of the VARIANTS field description is similar to that of the REDEFINES clause in VAX COBOL and VAX DATATRIEVE.

VAXcluster

A highly integrated organization of VMS systems that communicate over a high-speed communications path. VAXclusters have all the functions of single node VMS systems, plus the ability to share CPU resources, queues, and disk storage. Like a single-node system, the VAXcluster organization provides a single security and management environment. Member nodes may share the same operating environment or serve specialized needs.

video attribute

A characteristic of a TDMS form that provides one or more of the following special visual effects to an area of a form:

- Reverse video
- Bolding
- Blinking
- Underlining
- Double-height characters
- Double-width characters

view

A subset of an Rdb/VMS database that includes any combination of fields and records from a single relation or from different relations. You form a view using a record selection expression. To the user, the results look like a single relation.

In DATATRIEVE, the term view is used to refer to a view domain.

view domain

A special type of DATATRIEVE domain that lets you select some (or all) fields in some (or all) records from one or more domains. You can use a view domain to refer to fields and field values in the same or different domains without having to duplicate the data in those domains.

virtual field

A field that does not occupy any space in storage. The DATATRIEVE COMPUTED BY clause defines a virtual field. The value of the field is calculated when you access it with a DATATRIEVE statement.

VMS

The operating system on a VAX computer.

VMS image

See image.

VMS process

See process.

VMS user

A person or account authorized by a VMS system manager to access a VMS system. A VMS user is assigned a user name, a password, a user identification code (UIC), a default directory, a default command language, quotas, limits, and privileges.

wildcard character

A symbol, such as the asterisk or percent sign, that you use in place of all or part of a file specification.

workspace

In ACMS, a buffer used to save variable context between steps and tasks, whose description is stored in the CDD. A workspace can also hold application parameters and status information. Workspaces are passed to step procedures as parameters. ACMS provides record descriptions for three task workspaces, which are referred to as the system workspaces.

See also group workspace, system workspace, task workspace, and user workspace.

workspace symbol module

An object module, produced as a result of building a task group definition, that contains a main routine and debug symbol table used by the ACMS Task Debugger to examine workspaces. The object module must be converted into an executable image by the LINK command before the Task Debugger can use it.

Master Index

Master Index Book List

ACADG *VAX ACMS Application Definition Guide*

ACADR *VAX ACMS Application Definition Reference Manual*

ACAMG *VAX ACMS Application Management Guide*

ACAPG *VAX ACMS Application Programming Guide*

ACDAP *Developing Applications with VAX ACMS*

ACDPG *VAX ACMS Development Pocket Guide*

ACDSG *VAX ACMS Application Design Guide*

ACTDG *VAX ACMS Task Definition Guide*

ACTUG *VAX ACMS Terminal User's Guide*

CDDDL *VAX Common Data Dictionary Data Definition Language Reference Manual*

CDDUG *VAX Common Data Dictionary User's Guide*

CDUTL *VAX Common Data Dictionary Utilities Reference Manual*

DBDBA *VAX DBMS Database Administration Reference Manual*

DBDGD *VAX DBMS Database Design Guide*

DBDSG *VAX DBMS Database Security Guide*

DBFDM *VAX DBMS FDML Reference Manual*

DBIDA *VAX DBMS Introduction to Database Administration*

DBIDM *VAX DBMS Introduction to Database Manipulation*

DBLGD *VAX DBMS Load/Unload Guide*

DBMPG *VAX DBMS Maintenance and Performance Guide*

DBPRG *VAX DBMS Programming Guide*

DBPRM *VAX DBMS Programming Reference Manual*

DTGGR *VAX DATATRIEVE Guide to Using Graphics*

DTGPG *VAX DATATRIEVE Guide to Programming and Customizing*

DTHB *VAX DATATRIEVE Handbook*

DTREF *VAX DATATRIEVE Reference Manual*

DTRPT *VAX DATATRIEVE Guide to Writing Reports*

DTUG *VAX DATATRIEVE User's Guide*

RDDBD *VAX Rdb/VMS Guide to Database Design and Definition*

RDGAM *VAX Rdb/VMS Guide to Database Administration and Maintenance*

RDGDM *VAX Rdb/VMS Guide to Data Manipulation*

RDGP *VAX Rdb/VMS Guide to Programming*

RDREF *VAX Rdb/VMS Reference Manual*

TDAPG *VAX TDMS Application Programming Manual*

TDFRM *VAX TDMS Forms Manual*

TDREQ *VAX TDMS Request Manual*

TDSAM *VAX TDMS Sample Application Manual*

TDSUP *VAX TDMS V1.4 Documentation Supplement*

Master Index 3

In this index, a page number followed by a "t" indicates a table reference. A page number followed by an "f" indicates a figure reference. A page number followed by an "e" indicates an example reference.

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W