AA–N038F–TE

VAX Rdb/VMS

Guide to Using RDO, RDBPRE, and RDML

December 1990

This manual provides information about data manipulation and programming with relational databases using the following VAX Rdb/VMS interfaces: interactive RDO, Callable RDO, and the RDBPRE and RDML preprocessors.

Revision/Update Information:	This manual is a revision and supersedes previous versions.
Operating System:	VMS
Software Version:	VAX Rdb/VMS Version 4.0

digital equipment corporation
maynard, massachusetts

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

2 Accessing a Database and Using Transactions

3 Using Record Selection Expressions

14 Using the COBOL Program Environment

19 Using the Callable RDO Program Environment

Figures

Tables

Preface

VAX Rdb/VMS software, referred to as Rdb/VMS in this manual, is a general purpose database management system based on the relational data model.

Purpose of This Manual

This manual describes how to access, retrieve, and update data stored in an Rdb/VMS database, either interactively or using application programs written in high-level programming languages such as BASIC, C, COBOL, FORTRAN, and Pascal.

Note *SQL (structured query language), an industry-standard interface, is also included with VAX Rdb/VMS, and can be used to perform the complete range of operations described in this manual. The* VAX Rdb/VMS Guide to Using SQL *and the* VAX Rdb/VMS SQL Reference Manual *contain detailed information and examples.*

Intended Audience

This manual is intended for all users of Rdb/VMS who need to perform data manipulation operations using the RDO utility or Callable RDO in programs, and for experienced programmers who are familiar with BASIC, C, COBOL, FORTRAN, or Pascal and who are also familiar with Rdb/VMS data manipulation and data definition statements.

To get the most out of this manual, you should be familiar with data processing procedures, basic database management concepts and terminology, and the VMS Run-Time Library, system services, and operating system. Specifically, you should be familiar with the concepts and techniques described in the *VAX Rdb/VMS Introduction and Master Index*, the *Guide to Using VMS*, and the *Guide to VMS Programming Resources*.

Operating System Information

Information about the versions of the operating system and related software that are compatible with this version of Rdb/VMS is included with the Rdb/VMS media in the *VAX Rdb/VMS Installation Guide*.

For information on the compatibility of other software products with this version of Rdb/VMS, refer to the System Support Addendum (SSA) that comes with the Software Product Description (SPD). You can use the SPD/SSA to verify which versions of your operating system are compatible with this version of Rdb/VMS.

Structure

This manual contains the following chapters and appendix:

Chapter 1	Provides an informal introduction to the concepts of data organization and manipulation, and the relational database model. It demonstrates how to use RDO and describes how Rdb/VMS statements can be used in programs.
Chapter 2	Introduces Rdb/VMS data manipulation statements. It demonstrates how to access a database and explains how to use transactions.
Chapter 3	Describes RDO and record selection expressions (RSEs).
Chapter 4	Describes the process of retrieving data from one or more relations and joining the relations.
Chapter 5	Describes how to define views and how to use them in queries.
Chapter 6	Describes the Rdb/VMS data manipulation statements used to update databases.
Chapter 7	Describes the programming interfaces you can use to access an Rdb/VMS database and describes how to create a program prototype.
Chapter 8	Describes how to select and use host language data types that are compatible with Rdb/VMS data types.
Chapter 9	Describes how to structure, design, and develop Rdb/VMS application programs.
Chapter 10	Describes how to handle Rdb/VMS run-time errors.
Chapter 11	Describes how to preprocess, link, run, and debug Rdb/VMS application programs.
Chapter 12	Describes how to use the RDBPRE preprocessor interface.
Chapter 13	Describes how to use Rdb/VMS statements in BASIC preprocessed application programs.
Chapter 14	Describes how to use Rdb/VMS statements in COBOL preprocessed application programs.

Chapter 15	Describes how to use Rdb/VMS statements in FORTRAN preprocessed application programs.
Chapter 16	Describes how to use the RDML preprocessor interface.
Chapter 17	Describes how to use RDML statements in C preprocessed application programs.
Chapter 18	Describes how to use RDML statements in Pascal preprocessed application programs.
Chapter 19	Describes how to use RDO statements in Callable RDO Rdb/VMS application programs.
Appendix A	Lists common Rdb/VMS symbolic error codes.

Related Manuals

The other manuals in the Rdb/VMS documentation set are:

- *VAX Rdb/VMS Introduction and Master Index*

  Introduces Rdb/VMS and explains major terms and concepts. Includes a glossary, a directory of Rdb/VMS documentation, and a master index that combines entries from all the Rdb/VMS manuals.

- *VAX Rdb/VMS Guide to Database Design and Definition*

  Explains how to design a logical database and how to translate that design into a physical database using Rdb/VMS data definition statements.

- *VAX Rdb/VMS Guide to Database Maintenance and Performance*

  Provides guidelines for maintaining good database performance and explains how to use the database maintenance utilities to perform backup and recovery operations, restore journals, and analyze the database.

- *VAX Rdb/VMS Guide to Database Tuning*

  Introduces the concept of tuning, and explores how tuning the system, the database, and the application can affect database performance. Outlines a series of steps to follow in identifying, analyzing, isolating, and solving a performance problem, and in monitoring the resulting solution. Includes a set of decision trees that provide an organized approach to solving some common database tuning problems.

- *VAX Rdb/VMS Guide to Using SQL*

  Introduces the Rdb/VMS SQL (structured query language) interface, and shows how to retrieve, store, and update data interactively and through application programs.

- *VAX Rdb/VMS Guide to Using SQL/Services*

  Describes how to develop application programs that use SQL/Services, a client/server software component of Rdb/VMS that allows programs, from various remote computers running the Macintosh, MS-DOS, OS/2, ULTRIX, ULTRIX for RISC, or VMS operating systems, to access Rdb/VMS or VIDA databases on a VMS server system.

- *VAX Rdb/VMS Guide to Distributed Transactions*

  Describes the two-phase commit protocol and distributed transactions, explains how to start and complete distributed transactions using SQL, RDBPRE, and RDML, and how to recover from unresolved transactions using RMU commands.

- *VAX Rdb/VMS SQL Reference Manual*

  Provides reference material and a complete description of the statements, the interactive, dynamic, and module language interfaces, and the syntax for SQL, the structured query language interface for Rdb/VMS.

- *VAX Rdb/VMS SQL Quick Reference Guide*

  Summarizes the information in the *VAX Rdb/VMS SQL Reference Manual*.

- *VAX Rdb/VMS RDO and RMU Reference Manual*

  Provides reference material and a complete description of the statements and syntax of the Rdb/VMS Relational Database Operator (RDO) interface and the commands of the Rdb/VMS Management Utility (RMU).

- *RDML Reference Manual*

  Describes the syntax and use of the Relational Data Manipulation Language (RDML), which can be embedded in VAX C or VAX Pascal programs to access Rdb/VMS or Rdb/ELN databases.

- *VAX Rdb/VMS Installation Guide*

  Describes how to install Rdb/VMS.

- *VAX Rdb/VMS Release Notes*

  Describes new features, problems and problems fixed, restrictions, and other information related to the current release of Rdb/VMS. Contains information about SQL and other Rdb/VMS interfaces and utilities.

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the RETURN key at the end of a line of input.

Often in examples the prompts are not shown. Generally, they are shown where it is important to depict an interactive sequence exactly; otherwise, they are omitted in order to focus full attention on the statements or commands themselves.

This section explains the conventions used in this manual:

CTRL/x	This symbol in examples tells you to press the CTRL (control) key and hold it down while pressing the specified letter key.
RETURN	This symbol in examples indicates the RETURN key.
TAB	This symbol in examples indicates the TAB key.
. . .	A vertical ellipsis in an example means that information not directly related to the example has been omitted.
. . .	A horizontal ellipsis in statements or commands means that parts of the statement or command not directly related to the example have been omitted.
e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
Rdb/VMS statement	Within the context of this manual, *Rdb/VMS statement* is a generic term that includes RDO, RDML, and RDBPRE statements, but does not include SQL statements.
Color	In printed manuals, color in examples shows user input.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
$	The dollar sign represents the DIGITAL Command Language prompt. This symbol indicates that the DCL interpreter is ready for input.

References to Products

The VAX Rdb/VMS documentation set to which this manual belongs often refers to the following Digital products by their abbreviated names:

- DECdecision software is referred to as DECdecision.
- DEC RdbExpert for VMS software is referred to as RdbExpert.
- DECtrace for VMS software is referred to as DECtrace.

- The SQL interface to VAX Rdb/VMS is referred to as SQL. The SQL interface is Digital Equipment Corporation's implementation of the SQL standard ANSI X3.135-1989, ISO 9075:1989, commonly referred to as ANSI/ISO.
- VAX BASIC software is referred to as BASIC.
- VAX C software is referred to as C.
- VAX CDD/Plus software is referred to as CDD/Plus, the data dictionary, or the dictionary.
- VAX COBOL software is referred to as COBOL.
- VAX DATATRIEVE software is referred to as DATATRIEVE.
- VAX FORTRAN software is referred to as FORTRAN.
- VAX Pascal software is referred to as Pascal.
- VAX RALLY software is referred to as RALLY.
- VAX Rdb/VMS software is referred to as Rdb/VMS. Version 4.0 of VAX Rdb/VMS software is often referred to as V4.0.
- VAX TEAMDATA software is referred to as TEAMDATA.

Technical Changes and New Features

Many of the new features available in Rdb/VMS Version 4.0 are described in this manual. In addition, modifications to this manual have been made to reflect technical changes or to clarify or correct the documentation. For a list of all the new features in Rdb/VMS Version 4.0, see the *VAX Rdb/VMS Release Notes*; for a list of new RDO features, see the *VAX Rdb/VMS RDO and RMU Reference Manual*.

1

Introduction to VAX Rdb/VMS Data Manipulation

This chapter introduces Rdb/VMS concepts of data organization and data manipulation. It provides a brief overview of the relational model, and describes how to use Rdb/VMS statements with interactive RDO and in application programs. This chapter is divided into the following main sections:

- What Is a Relational Database?

  Introduces the concepts that are the basis for relational database management systems.

- Using RDO

  Explains how to use the Relational Database Operator (RDO) utility, the interactive environment for Rdb/VMS.

- Using the Sample Database

  Introduces the sample personnel database that you can create in two forms, single-file and multifile, and lists the files used in creating both forms. The sample personnel database is used throughout the Rdb/VMS documentation in examples.

- Using Rdb/VMS Statements in Programs

  Shows how to include RDO statements in high-level language programs. Programmers can read this section as an introduction to programming with Rdb/VMS.

- Internationalization Support

  Describes features that enhance the usability of Rdb/VMS in environments where the users' primary language is not English or where the data stored in the database is not in English.

If you have not defined RDO as a global symbol, type the following symbol definition. It is suggested that you include this definition in your LOGIN.COM file.

```
$ RDO :== RUN SYS$SYSTEM:RDO
```

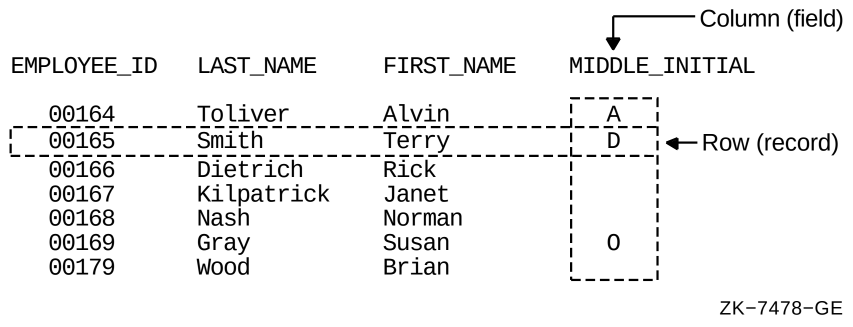
1.1 What Is a Relational Database?

In a relational database, data resides in two-dimensional tables known as **relations**. A relation consists of rows and columns. Each row contains a record (a set of data items). The columns, which usually have names, divide each row into a set of fields. For a single field within a record, there is only one data item.

Note *In this manual, the terms "record" and "field" are normally used rather than "row" and "column," because the former terms reflect the traditional RDO terminology, whereas the latter terms reflect SQL terminology.*

Figure 1–1 represents a typical Rdb/VMS relation that shows employee information. This relation is a subset of the sample personnel database (see Section 1.3).

Figure 1–1 A Typical Rdb/VMS Relation in Table Form



EMPLOYEE_ID	LAST_NAME	FIRST_NAME	MIDDLE_INITIAL
00164	Toliver	Alvin	A
00165	Smith	Terry	D
00166	Dietrich	Rick	
00167	Kilpatrick	Janet	
00168	Nash	Norman	
00169	Gray	Susan	O
00179	Wood	Brian	

Column (field)
Row (record)
ZK−7478−GE

In this relation, each field represents a particular item of data for each employee. Each record represents the data on 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. (Like relations, fields also have names.)

For example, assume you wished to find the employee identification number (employee ID) for Terry Smith and to display his first name, last name, and ID number. You need to enter a query specifying the EMPLOYEES relation, identifying the record you want as one where the LAST_NAME field is "Smith" and the FIRST_NAME field is "Terry", and naming the fields to be displayed. The result of this query might be as follows:

```
FIRST_NAME   LAST_NAME      EMPLOYEE_ID
Terry        Smith          00165
```

If you are familiar with VAX COBOL or VAX DATATRIEVE, you have probably used a COBOL file description or a DATATRIEVE record definition. A record definition with no group fields or OCCURS clauses is similar to a relation. An Rdb/VMS record, however, differs from a COBOL record in two ways:

- An Rdb/VMS relation cannot have repeating groups (lists). A maximum of one data item occupies a single named field in the record.
- An Rdb/VMS record cannot have group fields. A name within an Rdb/VMS relation refers to only one field.

Without repeating groups and group fields, the structure of the database is simplified so you may easily access each data item.

1.1.1 Single-File and Multifile Databases

A relational database can reside in a single file or in multiple files. In a single-file database, the actual data, the **metadata** (information about the data, such as relation and field definitions), and Rdb/VMS system information are all stored in one file (the database root file), which has a default file type of RDB. In a multifile database, the metadata and Rdb/VMS system information are stored in the RDB file, and the actual data is stored in one or more storage area files, which have a default file type of RDA.

Single-file databases are easier to design and define. Multifile databases require careful design, but can increase the database capacity and offer performance improvements. Moreover, some Rdb/VMS features, such as hashed indexes (discussed in Chapter 2), are available only in a multifile database. For detailed information on multifile databases, see the *VAX Rdb/VMS Guide to Database Design and Definition*.

1.1.2 Using Normalization to Eliminate Data Redundancy

There is no way to represent repeating groups of data items in an Rdb/VMS relation; only one data item can occupy an intersection of a record and field. Therefore, if you wanted to store information about five previous jobs for an employee, you would have to repeat the name, address, identification number, and other employee information five times. There would no longer be a one-to-one correspondence between the number of records in the relation and number of employees in the company.

If you stored all the information that might be relevant to employees in one relation, this would sometimes require that you store the same data in more than one place. This redundancy of data has two disadvantages:

- It wastes space in the database.
- It makes updating information difficult. For example, if you store the salary ranges for five previous jobs for an employee in the EMPLOYEES relation, you must find and change all the occurrences whenever the salary ranges change.

  To illustrate: if every EMPLOYEES record contained a record for each job the person held, and each record contained the minimum salary for that job, then if the minimum salary for an Associate Programmer was raised to $17,000, that information would have to be changed in the record of every employee who is an Associate Programmer. If you miss some, the database is no longer consistent. On the other hand, if the minimum salary for a job is stored only in a relation called JOBS, you would have to make the change only in the JOBS record for the Associate Programmer job.

A process known as normalization solves these two problems. **Normalization** ensures that the database keeps separate concepts logically separate and eliminates data redundancy. 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 (if you want an employee's job history, for instance), the database allows you to create temporary relationships by joining relations together. Rdb/VMS works best with well-designed, normalized databases.

1.2 Using RDO

You can define and access an Rdb/VMS database using the Relational Database Operator (RDO) utility. When you run RDO and type statements at the RDO> prompt, Rdb/VMS executes the statements immediately. This section shows you how to start using RDO and gives a brief introduction to elements of the RDO utility.

Note *SQL (structured query language), an industry-standard interface, is also included with VAX Rdb/VMS. Although RDO is used in the examples in this manual, you can also perform all the operations using SQL. The* VAX Rdb/VMS Guide to Using SQL *and the* VAX Rdb/VMS SQL Reference Manual *contain detailed information and examples.*

1.2.1 Beginning an RDO Session

To invoke RDO, type the following at the DIGITAL Command Language (DCL) prompt, or use a command symbol that equates to the following:

```
$ RUN SYS$SYSTEM:RDO
```

RDO responds with the RDO> prompt. Prompts help you keep track of your status during an interactive RDO session. The RDO prompts are:

RDO>	RDO command level prompt. This prompt tells you that you are typing commands to RDO and may enter any RDO statement.
cont>	The statement continuation prompt. This prompt indicates that you have not yet entered a complete statement.

RDO incorporates many features to make working with Rdb/VMS easy. These features include:

- HELP statement

  Provides information about Rdb/VMS statements and concepts.

- SHOW statement

  Displays information about the database, including the names and attributes of fields, the structure of relations, and the definitions of indexes, constraints, and triggers. The SHOW statement also displays information about the version of Rdb/VMS you are using.

- SET statement

  Specifies certain characteristics and defaults for an RDO session.

- Command recall

  Lets you recall up to the last 20 RDO statements you issued. You use the up arrow and down arrow keys, just as with command recall at the DCL level. You can also edit any recalled statement.

- Indirect command file

  Lets you store RDO statements and execute them later by using the at sign (@). The default file type is RDO.

- DCL command invocation

  Lets you access DCL commands from RDO by using the dollar sign ($). Thus, for example, you do not have to leave your RDO session to answer mail or to see a directory listing.

- EDIT

  Calls a VMS editor (VAX EDT by default, although you can specify the VAX Text Processing Utility (VAXTPU)). Type EDIT * or EDIT followed by an integer to edit a number of your previous statements. You also can use EDT or VAXTPU from inside RDO to insert successful RDO statements into command files and programs.

- RDOINI.RDO

  A startup file that you can create. When you enter RDO, the commands in your RDOINI.RDO file are automatically executed. You may create RDOINI files in many directories, or define a logical name RDOINI to point to a central startup file.

1.2.2 Getting Online Help in RDO

If you need an explanation of any RDO statement or concept while using RDO, type HELP at the RDO> prompt to see a list of available topics, or type HELP and the name of a topic:

```
RDO> HELP DEFINE_FIELD
```

The help function contains several levels. For example, if you type HELP DEFINE_FIELD, you will see a brief description of the DEFINE FIELD statement, an example, and a set of choices, including one called Format. The Format choice shows the syntax of the DEFINE FIELD statement.

Note that many topics contain underscores (for example, those starting with CHANGE, DEFINE, and DELETE). This means, for example, that you will receive an "error" if you type HELP DEFINE FIELD; however, this design is necessary so that all the format (syntax) diagrams will display correctly, and it also has the advantage of including more information in the top-level help display.

Once you have located the relevant piece of information in the help files, you can exit from help by pressing CTRL/Z or by pressing RETURN until you come back to the RDO> prompt.

1.2.3 Using Record Selection Expressions

The following example shows a record selection expression followed by a request to display the selected records:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = 'Smith' AND
  E.FIRST_NAME = 'Terry'
    PRINT
      E.FIRST_NAME,
      E.LAST_NAME,
      E.EMPLOYEE_ID
END_FOR
```

In this example, WITH E.LAST_NAME = 'Smith' AND E.FIRST_NAME = 'Terry' is a record selection expression.

A **record selection expression** (RSE) is a phrase that defines specific conditions that individual records must meet before Rdb/VMS includes them in a record stream. The **record stream** is the group of records from one or more relations returned by Rdb/VMS to the interactive user or the application program. The RSE in a data manipulation statement determines which records are included in the record stream. In this case, only records in which the employee's last name is Smith and the first name is Terry are included in the record stream. You can include all the records of a relation, or you can restrict the record stream to a selected group of records.

Once you form the stream, you can enter statements to display, store, modify, or erase the data in the stream, one record at a time. To display the results of an RSE, use the PRINT statement. The PRINT statement uses values from the record stream that you identify in the RSE.

The character E in the expression E IN EMPLOYEES is a **context variable**, a temporary name that you choose to associate with a specific relation (in this case, EMPLOYEES). RDO requires the use of context variables in most data manipulation statements. You can choose almost any arbitrary string as a context variable; however, for convenience and clarity, it is recommended that a context variable be short. Thus, a context variable is usually a single letter (the first letter in the relation name), or some abbreviation that is easy to associate with the relation name.

The context variable E in the preceding example lets you refer to the EMPLOYEES relation specifically in the RSE and in the PRINT statement. Context variables are particularly important when you are working with more than one relation. If two relations have fields with the same name, the context variables enable you to specify the fields explicitly.

1.2.4 Using Multiline Statements in RDO

RDO can read enough lines in a multiline statement to detect a syntactically complete statement. If you end each line of an RSE with a keyword that belongs with the next line, RDO will wait for the entire sequence of statement lines before it executes them, as in the following example:

```
RDO> PRINT TOTAL SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY CROSS
cont>  JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
cont>  JH.JOB_CODE = "MENG" AND
cont>  JH.JOB_END MISSING
              685094.00
RDO>
```

In this example, which totals the salaries of all employees who currently have MENG as their job code, RDO issues the RDO> prompt on the first line and the cont> prompt on the remaining lines.

You can also end each line of a multiline statement with the hyphen (-) continuation character to ensure that RDO reads the whole statement before execution. The continuation character must be the last character on the line to be continued. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for a full explanation of the input format that RDO accepts.

1.2.5 Exiting from RDO

You end an RDO session by typing EXIT or pressing CTRL/Z. Either method ends a session and normally returns you to the DCL prompt ($). For example:

```
RDO> EXIT
$
```

If you have made updates to the database or changed data definitions without finishing the transaction, you cannot immediately exit from RDO. If you try to exit, RDO responds that there are uncommitted changes, and it asks if you would like the chance to commit these changes. If you respond YES, you are returned to the RDO> prompt, and you can then type COMMIT, ROLLBACK, or any other RDO statement. (For an explanation of COMMIT and ROLLBACK, see Section 2.3.9.) If you respond NO, you will leave the RDO session and return to the DCL prompt without saving any changes you may have made to the database.

Try the following statements to see how these features work. (The text does not show the output.)

```
RDO> HELP
RDO> HELP SET
RDO> HELP RDOINI
RDO> HELP DEFINE DATABASE
(Press CTRL/Z to leave the help facility)
RDO> $ DIRECTORY
RDO> $ MAIL
(Type EXIT to leave the Mail utility)
RDO> EDIT *
(Exit from the editor, then exit from RDO)
```

1.3 Using the Sample Database

The examples throughout this guide use a sample personnel database that you can build using files supplied with the Rdb/VMS installation kit; these files are listed in Table 1–1. The database can actually be created in two forms: a single-file form (PERSONNEL) and a multifile form (MF_PERSONNEL). The command files to build copies of both forms of the database are located in the directory RDM$DEMO.

Table 1–1 Files to Create Sample Database

File Name	Explanation
PERSONNEL.COM	Builds a single-file or multifile version of the sample personnel database; you can use RDO or SQL definitions and you can define the database by file name or data dictionary path name. For the RDO single-file version, the procedure invokes RDO command files to define global fields, relations, views, constraints, sorted indexes, and triggers. In addition, it invokes programs to store most of the data. For the RDO multifile version, the procedure invokes RDO command files to define global fields, relations, views, constraints, indexes (both sorted and hashed), and to spread relations and indexes across multiple files. In addition, it invokes programs to store most of the data.
BUILDPERS_RDO.RDO	Defines global fields, relations, and views, and stores some data.
MF_BUILDPERS_ RDO.RDO	Defines global fields, relations, hashed indexes, storage maps, views, and one sorted index, and stores some data.
RDO_DEFINE_ STORAGE.RDO	Defines storage areas for the multifile database.
PERSONNEL_ INDEXES_RDO.RDO	Defines the indexes (sorted) for the single-file database.
MF_PERSONNEL_ INDEXES_RDO.RDO	Defines the remaining sorted indexes for the multifile database.
CONSTRAINTS_ RDO.RDO	Defines constraints.
TRIGGERS_RDO.RDO	Defines triggers.

The definitions of fields, relations, views, constraints, and triggers are the same for the single-file and multifile databases. The definitions for the multifile database also include hashed indexes, storage area files, and other structures associated with the multifile implementation.

1.3.1 Creating the Sample Database

You use a single command procedure (RDM$DEMO:PERSONNEL.COM) to create the database, and you can specify parameters when you invoke the procedure to specify certain options, such as whether you want the single-file or multifile version and whether you want the database created using RDO or SQL statements. The format of the command you enter to create the sample database is shown in the next example.

```
$ @RDM$DEMO:PERSONNEL interface-language database-form dictionary-use
```

The three parameters and their defaults are as follows:

- interface-language: SQL or RDO. Default: SQL.
- database-form: S (single-file) or M (multifile). Default: S.
- dictionary-use: CDD (use CDD/Plus dictionary) or NOCDD (do not use dictionary). Default: allow the user to choose.

The procedure also displays the approximate number of disk blocks that will be used and allows the user to exit the procedure; thus, you may wish to omit the dictionary-use parameter.

You may use upper case, lower case, or mixed case to specify the parameters. All parameters are optional; for example, to create a single-file database using SQL definitions and have a menu ask about dictionary use, you can simply enter:

```
$ @RDM$DEMO:PERSONNEL
```

However, if you want to specify the second or third parameter, you must also specify any preceding parameters. For example, to create a single-file database using RDO definitions, you must enter:

```
$ @RDM$DEMO:PERSONNEL RDO S
```

Regardless of the interface language used, PERSONNEL.COM creates a database named PERSONNEL.RDB if you are creating a single-file database, and it creates a database named MF_PERSONNEL.RDB (plus related storage area files) if you are creating a multifile database. Note also that you can use either the SQL or the RDO interface to work with the resulting database or databases, regardless of whether the database was created using SQL or RDO command files. There are differences between SQL-defined and RDO-defined databases. See the *VAX Rdb/VMS Introduction and Master Index* for more information.

Note *The log of the database definition statements used in creating the database is placed in a file called PERSONNEL.LOG in the same directory as the database files. The file is named PERSONNEL.LOG regardless of which options you specified or accepted as defaults (for example, regardless of whether you created a single-file or multifile database).*

1.4 Using Rdb/VMS Statements in Programs

As a programming tool, Rdb/VMS has the following advantages:

- The versatility of the data manipulation statements means that the database system itself can perform many of the tasks you once needed to code in a high-level language.
- The interactive environment, RDO, lets you create a prototype of your application before you start writing a program. With some modification, you can include the Rdb/VMS data manipulation statements in your programs.

Although the RSE you saw in Section 1.2.3 was processed by interactive RDO, Rdb/VMS is intended to be used in programs. Rdb/VMS includes a set of preprocessors that let you include data manipulation statements in programs, as if they were part of the language. A preprocessor translates the Rdb/VMS statements into subroutine calls and other host language constructs.

RDO provides an EDIT statement that makes developing these programs easy. Most often, you will use RDO to test queries and other data manipulation statements to make sure they produce the desired results. The EDIT statement lets you modify a statement you previously entered in RDO.

Rdb/VMS saves that statement in an editing buffer, so you may use a VMS text editor to change any portion of the editing buffer. Use the EDIT statement to open the edit buffer. You may repeat this process as many times as you wish, editing up to 20 of your previous statements. When you have the desired results, save the query by issuing a WRITE or EXIT command. You may then incorporate the query into your high-level language program.

The following examples show how RDO statements can be used in a program. Example 1–1 is a COBOL program that performs a store operation. This program reads the values for the database from the data file and stores them.

Example 1–1 COBOL Program Performing Store Operation

```
IDENTIFICATION DIVISION.
PROGRAM-ID.              STORE-REC.
*
* First, identify the input data file.  The program will
* read this file and store its records in the database.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT EMPLOYEES-FILE  ASSIGN TO "EMP.DAT"
        ORGANIZATION IS SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL.
*
* Instead of an explicit declaration, you can declare the
* record structure by copying a definition from the
* data dictionary. Then you declare the database
* file to the COBOL preprocessor with the DATABASE
* statement.
*
DATA DIVISION.
FILE SECTION.
FD EMPLOYEES-FILE.

        COPY "CDD$TOP.FORESTER.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
              FROM DICITONARY.

WORKING-STORAGE SECTION.

         &RDB&  INVOKE DATABASE FILENAME 'PERSONNEL'

PROCEDURE DIVISION.
*
* The PROCEDURE DIVISION consists simply of a database
* transaction in a loop.  Note the three steps:
*
* 1. Open the input file and start a transaction.
*    The RESERVING clause locks other users out of the
*    EMPLOYEES relation.
*
BEGIN.
        OPEN INPUT EMPLOYEES-FILE.
          &RDB&  START_TRANSACTION READ_WRITE
-          RESERVING EMPLOYEES FOR EXCLUSIVE WRITE.
*
* 2.  Read the file one record at a time and store
*     fields from the file into fields in the database relation:
*
```

(continued on next page)

Example 1–1 (Cont.) COBOL Program Performing Store Operation

```
READ-EMPLOYEES.
        READ EMPLOYEES-FILE AT END GO TO STORE-DONE.
         &RDB&  STORE E IN EMPLOYEES
-         USING
-           E.EMPLOYEE_ID = id;
-           E.LAST_NAME = LAST_NAME;
-           E.FIRST_NAME = FIRST_NAME;
-           E.MIDDLE_INITIAL = INITIAL;
-           E.ADDRESS = ADDRESS;
-           E.CITY = CITY;
-           E.STATE = STATE;
-           E.POSTAL_CODE = ZIP;
-       END_STORE
        GO TO READ-EMPLOYEES.
*
* 3.  Commit the transaction.  This makes the storage
*     operation complete.  The EXCLUSIVE share mode on the
*     relation is released. The FINISH statement tells
*     Rdb/VMS that you are done working with the
*     database.
*
STORE-DONE.
         &RDB&  COMMIT
         &RDB&  FINISH
        CLOSE EMPLOYEES-FILE.
        STOP RUN.
```

Example 1–2 is a COBOL program fragment that retrieves database values and assigns them to program variables. This example shows how to convert the RDO PRINT statement into a GET statement. The GET statement retrieves a value from the database and assigns it to a variable in the program. The GET statement uses the same kind of RSE as the PRINT statement. VAX BASIC, VAX COBOL and VAX FORTRAN programs use the RDBPRE preprocessor. Languages supported by the Relational Data Manipulation Language (RDML) preprocessor (VAX C and VAX Pascal) use a simple host language assignment statement instead.

Example 1–2 COBOL Program Retrieving Database Values

```
       DISPLAY "FIRST_NAME     LAST_NAME      id"

&RDB&  FOR E IN EMPLOYEES
       &RDB&  GET FIRST = E.FIRST_NAME;
              &RDB&   LAST = E.LAST_NAME;
              &RDB&   id = E.EMPLOYEE_ID:
       &RDB&  END_GET
       DISPLAY FIRST, "         ", LAST, "  ", id

&RDB&  END_FOR
```

1.5 Internationalization Support

Rdb/VMS provides several options that are useful when the data in the database is not in English or when the users' primary language is not English. These options fall into two categories:

- Statements to control the format of data for input and display
- Collating sequence specification (to control sorting and comparisons)

1.5.1 Controlling Input and Display Format

You can enter statements to modify the input and display format for the following:

- Radix point character
- Date and time format
- Language used for various input and displays, such as day names, month names, and so on.

The SET and SHOW statements related to these features are documented in the *VAX Rdb/VMS RDO and RMU Reference Manual*.

1.5.2 Specifying Collating Sequence

By default, Rdb/VMS uses the ASCII collating sequence for all sorting and Boolean operations; however, you can override this default by specifying one of the following:

- One of the language-specific collating sequences supplied by the VMS National Character Set (NCS) utility
- A user-defined collating sequence using the NCS utility

You can specify collating sequences for particular global fields. The collating sequence determines how records are sorted when the field is used as a sort key. The collating sequence also determines the behavior of Boolean operations that compare two fields or a field with a literal value. See Section 3.5 for a discussion of the behavior of some specific relational operators with non-English collating sequences.

The following list describes the RDO statements that let you specify collating sequences. For complete reference information on these statements, including syntax diagrams, see the *VAX Rdb/VMS RDO and RMU Reference Manual*.

- DEFINE COLLATING_SEQUENCE

  Allows you to specify a collating sequence that has been defined using the NCS utility. You must first identify a collating sequence using the DEFINE COLLATING_SEQUENCE statement before you use any of the remaining statements in this list.

- SHOW COLLATING_SEQUENCES

  Displays the collating sequence for the invoked database.

- DEFINE FIELD . . . COLLATING_SEQUENCE

  Specifies a collating sequence for a new global field.

- CHANGE FIELD . . . COLLATING_SEQUENCE

  Specifies a new collating sequence for a global field.

- DEFINE DATABASE . . . COLLATING_SEQUENCE

  Specifies a collating sequence that will be used for all fields in the database.

- IMPORT . . . COLLATING_SEQUENCE

  Specifies a collating sequence that will be used for all fields in the database.

Note that you cannot explicitly specify a collating sequence for a local field. You can define collating sequences only for databases and global fields, not for local fields. If you define a local field using a global field, however, the local field inherits any collating sequence you specify for the global field.

2 Accessing a Database and Using Transactions

This chapter shows you how to access a database and manipulate data using RDO. It describes how to use the:

- INVOKE DATABASE statement to tell RDO which database(s) you want to use
- START_TRANSACTION statement to specify how and when transactions affect the database
- COMMIT or ROLLBACK statement to end a transaction

2.1 Invoking a Database

Before you can access data managed by Rdb/VMS, you must name the database or databases you want to use with the INVOKE DATABASE statement.

Rdb/VMS stores definitions of database elements in the database file itself and, optionally, in the data dictionary if VAX CDD/Plus is installed. You can invoke the database by naming either its VMS file specification or its dictionary path name. If you intend only to retrieve or update the data itself, access the database by file name. If you intend to change data definitions, access the database by using the dictionary path name. When you access the database using the path name, any changes you make to database data definitions are entered in both the dictionary and the database; however, when you access the database by file name, data definition changes are made *only* in the database.

2.1.1 Accessing the Database by File Name

You can access an Rdb/VMS database by entering a file specification. If you omit parts of the file specification, standard VMS defaulting applies. For example, the following statements invoke the database PERSONNEL.RDB by file name:

```
RDO> INVOKE DATABASE FILENAME 'PERSONNEL'

RDO> INVOKE DATABASE FILENAME
cont> 'DISK1:[RDBDEMO.STAFF]PERSONNEL'
```

In the first INVOKE statement in the preceding example, the database file PERSONNEL.RDB is assumed to be in the current process default device and directory. In the second INVOKE statement, the user specifies a device and directory different from the current defaults.

You can also invoke a database by specifying a logical name that translates to a file specification. Using a logical name is especially recommended in a production environment.

2.1.2 Accessing the Database by Dictionary Path Name

You can access an Rdb/VMS database by specifying a CDD/Plus path name. You can enter a complete path name, or you can use a logical name for part or all of the path name. The following example shows a database being invoked by dictionary path name. The example also shows the use of the logical name CDD$DEFAULT for part of the path name.

```
$ DEFINE CDD$DEFAULT SYS$LOGIN_DEVICE:[SMITH.CDDPLUS]SMITH
    .
    .
    .
$ RDO
RDO> INVOKE DATABASE PATHNAME 'CDD$DEFAULT.PERSONNEL'
```

You can also use the SET DICTIONARY statement in RDO to change the current dictionary for this RDO session. Note that using the SET DICTIONARY statement in RDO does not change the equivalence name of the logical name CDD$DEFAULT.

Note *Place quotation marks around file names and dictionary path names to avoid ambiguity. The preprocessors require either single or double quotation marks around file names and path names. RDO accepts either quoted or unquoted file specifications. However, if you do not use quotation marks, Rdb/VMS may not interpret file names or path names correctly.*

2.1.3 Accessing the Database from a Remote Node

You can access an Rdb/VMS database from a remote node in a network using a full file specification in your INVOKE DATABASE statement. Assume you are logged in to node REM4 and the Rdb/VMS PERSONNEL database is located on the network node CENT in the DISK1:[COMPANY.STAFF] directory. The following INVOKE DATABASE statement gives you access to the PERSONNEL database on the remote node named CENT:

```
RDO> INVOKE DATABASE FILENAME
cont> 'CENT::DISK1:[COMPANY.STAFF]PERSONNEL'
```

Note that by default, the RDB$REMOTE account (supplied by Rdb/VMS) is used on the remote VAX node. (For details on RDB$REMOTE, see the *VAX Rdb/VMS Installation Guide*.) The RDB$REMOTE account is not used, however, if you specify an access control string in the database file specification or if you use a proxy account.

If you are using an access control string or a proxy account, you can improve performance over the network by modifying the LOGIN.COM procedure for the account specified or used. For example, if you define logical names for your databases, do so at the beginning of the LOGIN.COM file. Then include the following DCL command to bypass any other operations not necessary for network access:

```
$ IF 'F$MODE()' .EQS. "NETWORK" THEN $EXIT
```

2.1.4 Accessing Data

Once you have invoked a database, you can access the data in it. With a single-file database, there are two types of files Rdb/VMS uses. Use the DIRECTORY command at the DCL level to look at the files created when you typed @RDM$DEMO:PERSONNEL RDO to build the single-file form of the sample database:

```
$ DIRECTORY DISK1:[RDBDEMO.STAFF]PERSONNEL.*

PERSONNEL.RDB;1       PERSONNEL.SNP;1
```

Note *Rdb/VMS provides command files to build two sample databases: PERSONNEL, a single-file database, and MF_PERSONNEL, a multifile version of that database. For more information about those command files, see Section 1.3, especially Table 1–1.*

For detailed information on defining and using a multifile database, see the VAX Rdb/VMS Guide to Database Design and Definition.

The PERSONNEL.RDB file contains the following types of information:

- System information (used to maintain database integrity, locate related files, and so on)

- Data definitions (metadata) that describe the fields, relations, and indexes as they are defined in the database. You can think of metadata as a set of templates that describe the format, structure, and characteristics of database elements. The field, relation, and index definitions from the PERSONNEL database are examples of metadata.
- The actual data records (employee records, job history records, and so on) that were stored in the database after it was created.

The PERSONNEL.SNP file is called a snapshot file. It contains copies of data used for read-only transactions. (The snapshot file is not created for read-only storage areas). The section on using transactions explains the different transaction types, including the read-only transaction.

2.2 Using Transactions

Rdb/VMS allows many users access to a database at the same time, and it controls that access to avoid conflicts and data inconsistencies. Rdb/VMS, therefore, requires each user to identify a unit of database activity, called a transaction.

A **transaction** is a set of operations on the database that must complete as a unit or not complete at all. If, for example, you wanted to transfer an employee from one department to another, you would want the changes to all records for that employee to be made at the same time. If a software error or hardware failure occurred before all operations in several transactions completed, the database might show that the employee belonged to two departments or had two salaries (or belonged to no department or had no salary); thus the database would no longer be consistent. To avoid such inconsistencies, you include all such update tasks in a single transaction.

Transactions can have many characteristics, which you control with the START_TRANSACTION statement. A START_TRANSACTION statement signals the beginning of a transaction. The START_TRANSACTION statement options let you determine:

- Whether you want to work with the snapshot of the database or with the database itself
- Whether you intend to read or modify data in the relations
- What kind of access you allow other users to have to the database resources you are using
- When you want Rdb/VMS to consider specific conditions (called constraints) that must be satisfied before a record is stored or retrieved

In a START_TRANSACTION statement, you state or accept defaults for:

- The transaction mode you need. For example:
  - Read-only—if your transaction only retrieves data values from the database, but does not change them
  - Read/write—if your transaction changes values in the database
  - Batch-update—sometimes useful for initial loads of databases
- The names of the relations you want to access. (You can retrieve records from a single relation or from several relations joined together.)
- The control you want over the access other users have to the relations you reserve for your transactions.

When you access a specific record in a transaction, Rdb/VMS prevents other users from having certain kinds of access to that record by *locking* the record. The kind of record locking specified in a START_TRANSACTION statement begins when you enter a query. The records identified by the record selection expression (RSE) remain locked until you terminate your transaction. The record locks are then released and other users may access those records. Refer to the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for a complete description of how Rdb/VMS uses the locking mechanism.

The following sections explain how to use the read-only, read/write, and batch-update transactions and their options, and how these affect database performance.

With Rdb/VMS you can use distributed transactions, which allow you to access multiple database handles or multiple database management systems (for example, Rdb/VMS and VAX DBMS).

You access multiple database handles by attaching to:

- More than one Rdb/VMS database
- A single Rdb/VMS database more than once

Rdb/VMS uses the two-phase commit protocol, provided by DECdtm services, to ensure that every required operation is completed before a transaction is made permanent, even if the transaction attaches to databases that are on remote nodes. If one operation in a transaction cannot be completed, none of the operations is completed. This “all or nothing” approach guarantees that distributed databases remain logically consistent with one another.

For more information on distributed transactions, see the *VAX Rdb/VMS Guide to Distributed Transactions*.

2.3 Specifying the Transaction Mode

The START_TRANSACTION statement allows different types of access to relations in a database. You can establish restrictions on other users' access and declare your work intentions.

Every statement you enter with RDO must take place within the boundaries, or context, of a transaction, and the characteristics of the transaction determine the type of access you have to the database. Whether you plan to work with data definitions or actual data records, and whether you plan to modify the database or merely retrieve information, the type of statement you enter determines how Rdb/VMS lets you do that work.

If you do not enter a START_TRANSACTION statement to begin your work with a database, Rdb/VMS provides you with a default transaction depending on the first statement you issue in your interactive session. (Even if a transaction is started without a START_TRANSACTION statement, you must enter an explicit COMMIT or ROLLBACK statement before you can start another transaction.)

Rdb/VMS considers all statements to be one of two types and assigns a default transaction to each, depending on its type. The two types of statements are:

- Data manipulation statements

  You use data manipulation statements to access data. If you do not explicitly start a transaction, Rdb/VMS starts a read-only transaction for you. For example, if your first query after the INVOKE DATABASE statement displays data from the database, Rdb/VMS allows this task to execute. If, however, the first statement modifies or updates data, Rdb/VMS returns an error because a read/write transaction is required.

- Data definition language statements

  You use data definition statements to define, change, or delete database metadata. For example, if you need to change the data type of a field, or to define a new relation, you need update access to the database to make these changes. By default, Rdb/VMS starts a read/write transaction when you issue a data definition statement.

You should always issue a START_TRANSACTION statement to begin a transaction (as opposed to letting Rdb/VMS issue one by default), to prevent confusion or errors when you enter several statements before a COMMIT or ROLLBACK statement. (A COMMIT or ROLLBACK statement marks the end of the current transaction and the start of a new one. The COMMIT statement causes any changes you specified to be made to the database; the ROLLBACK statement returns the database to its pretransaction state—that is, no changes are made to the database.)

The format of the START_TRANSACTION statement and the meaning of the transactions are as follows:

```
RDO> START_TRANSACTION <transaction-mode>
```

- READ_ONLY

  All relations are available for data retrieval (unless blocked by another user's EXCLUSIVE share mode specification). Data values are those of the moment you entered your START_TRANSACTION statement. You do not see updates committed by other users while a read-only transaction is in effect. You may read any record in any relation to which you have authorized access through Rdb/VMS access rights.

- READ_WRITE

  All relations are available for data retrieval, and for addition, deletion, and modification of data and metadata (unless blocked by another user's EXCLUSIVE share mode specification). Rdb/VMS reserves each relation as you refer to it. You may update any record in any relation to which you have authorized access through Rdb/VMS access rights.

- BATCH_UPDATE

  Using a batch-update transaction reduces overhead in large load operations. To speed update operations, Rdb/VMS does not write any recovery-unit journal files in a batch-update transaction. Therefore, you cannot roll back a batch-update transaction; if the load fails, the database is corrupt, and you must create the database again. When you specify BATCH_UPDATE in your START_TRANSACTION statement, the load or update task has exclusive access to the entire database. It is efficient for loading the entire database for the first time, or for batching database updates in a data file that you intend to apply to the database at one time.

  However, because of the limitations of batch-update transactions, for most applications you should specify READ_WRITE RESERVING relation-name FOR EXCLUSIVE WRITE for loading data instead of specifying BATCH_UPDATE.

2.3.1 Read-Only Transactions

When you are updating values, you change them in the database file itself (the RDB file in a single-file database, the RDA file or files in a multifile database). If you only want to read values, however, they may be read from the snapshot file (SNP), unless the values are stored in a read-only storage area, in which case values are read directly from the storage area file because there is no corresponding snapshot file. This type of access is called a read-only transaction. When you use a read-only transaction, you read current versions of records not locked by any other user and previous versions of records that are locked. Because many transactions can share read locks that Rdb/VMS places on records in the snapshot file, your transaction does not conflict with others.

In many cases, it will not matter to you whether you are reading the “old” or “new” data in a record that is in the process of being updated. (The “old” data in this case is sometimes referred to as a “before-image” of the record.) However, if your transaction requires an absolutely current picture of the database, do not use a read-only transaction; instead, use a read/write transaction, and reserve the relations that you need to access by specifying EXCLUSIVE access.

Specify READ_ONLY in your START_TRANSACTION statement when you do not intend to add new records or to change existing values in the database, and when it is not essential to get the latest values of volatile data. You must specify READ_ONLY if you are accessing read-only storage areas. Note that snapshot files are not created for read-only storage areas. A read-only transaction minimizes Rdb/VMS overhead operations, and is thus the best choice when creating reports and performing queries of the database.

If your application modifies data in certain relations very infrequently, or not at all, you may improve your database performance by placing these relations in read-only storage areas in a multifile database. For more information on read-only storage areas, refer to the *VAX Rdb/VMS RDO and RMU Reference Manual*, the *VAX Rdb/VMS Guide to Database Design and Definition*, and the *VAX Rdb/VMS Guide to Database Maintenance and Performance*.

2.3.2 Read/Write Transactions

Specify READ_WRITE in your START_TRANSACTION statement when you want to be able to perform additions, deletions, or changes to the database (for example, using the STORE, ERASE, or MODIFY statements). When you need read/write access to the database, you can use several formats of the START_TRANSACTION statement.

In one format of the START_TRANSACTION statement, you merely specify READ_WRITE to start the read/write transaction to allow update operations in the database:

```
RDO> START_TRANSACTION READ_WRITE
```

This format does not name a specific relation or relations for database updates. Rdb/VMS reserves the relations as you name them in your statements and, depending on the type of operation you perform in your transaction, places write locks on selected records to complete an update task.

For example, the first data manipulation statements in your transaction might retrieve data from the EMPLOYEES relation. Rdb/VMS locks the records necessary to make an update. Later in the transaction you might modify values in selected records in the EMPLOYEES relation. Rdb/VMS, using only the necessary locks to complete the transaction, would promote the level of record locking.

In another format of the START_TRANSACTION statement for a read/write transaction, you name the relations you need to read and specify what you will allow other users to do when they access the same relations. To get the higher locking you need for certain read operations, specify READ_WRITE with the correct share mode for your transaction. For example:

```
RDO> START_TRANSACTION READ_WRITE RESERVING JOBS FOR EXCLUSIVE WRITE
```

See Section 2.3.4 for information on options relating to shared, protected, and exclusive access to data in a relation.

2.3.3 Batch-Update Transactions

You can reduce overhead in large load operations by using a batch-update transaction. To speed update operations, Rdb/VMS does not write to any journal files in batch-update mode. Therefore, you cannot roll back a batch-update transaction; if the load fails, you must create the database again. Consequently, it is usually preferable to specify READ_WRITE RESERVING relation-name FOR EXCLUSIVE WRITE to load data instead of specifying BATCH_UPDATE.

When you can specify BATCH_UPDATE in your START_TRANSACTION statement, the load or update task results in access to the entire database. Specifying BATCH_UPDATE thus requires that your transaction be the *only* transaction accessing the database. It is the most efficient choice when you are loading the entire database for the first time, or when you batch database updates in a data file that you intend to apply to the database at one time.

Because a batch-update transaction does not create before-images of changed records and because you cannot roll back a batch-update transaction, *you should create a backup copy* of the database using the VMS Backup utility before starting the transaction.

The following is a sample session that shows the effects of issuing a ROLLBACK statement. First, the user backs up the database:

```
$BACKUP/LOG PERSONNEL.* PERSBACKUP.BCK/SAVESET
%BACKUP-S-COPIED, copied DISK1:[CORP.DBS]PERSONNEL.RDB;1
%BACKUP-S-COPIED, copied DISK1:[CORP.DBS]PERSONNEL.SNP;1
```

Then, the user invokes RDO.

```
$RDO
RDO> INVOKE DATABASE FILENAME PERSONNEL
RDO> START_TRANSACTION BATCH_UPDATE
RDO> STORE E IN EMPLOYEES USING
cont> E.EMPLOYEE_ID = "15399";
cont> E.LAST_NAME = "North";
cont> E.FIRST_NAME = "Oscar";
cont> END_STORE
RDO> ! At this point, assume the user does not know that
RDO> ! rolling back a batch-update transaction will corrupt the
RDO> ! database. This user now enters a ROLLBACK statement:

RDO> ROLLBACK
%RDB-E-NOROLLBACK, no rollback is allowed with the recovery mechanism disabled
```

At this point, the user's batch-update transaction is still active:

```
RDO> SHOW TRANSACTION
All Transactions in Database with filename PERSONNEL
a read/write transaction is in progress
 - updates have been performed
 - transaction sequence number (TSN) is 152
 - snapshot space for TSNs less than 152 can be reclaimed
 - session ID number is 55
```

If you receive the RDB$_NOROLLBACK error during a batch-update transaction, you have two choices:

1. Manually undo any changes you made (or fix the problem you were having) and then commit the transaction. For example, if you stored a record, erase that record and then issue a COMMIT statement:

   ```
   RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "15399"
   cont> ERASE E
   cont> END_FOR
   RDO> COMMIT
   RDO> FINISH
   RDO> EXIT
   ```

   Or, if invalid input caused a constraint to fail, enter the correct (valid) data and then issue a COMMIT statement.

2. Exit the program or RDO session, which will corrupt the database.

The second option assumes that you made a backup copy of the database before starting the batch-update transaction. After restoring the database files from the backup file (default file type BCK), you can correct the situation that led to the error and then start the update program or RDO session again.

2.3.4 Reserving Options

The START_TRANSACTION statement lets you reserve different relations in the database for different types of access. By reserving just the relations you need and specifying the appropriate access, you can minimize system overhead. Every lock Rdb/VMS places on a database, relation, page, or index node reduces the lock resources available to other processes on your system, and also increases the possibility of input/output contention and deadlock. By specifying the relations you need and the required access in the START_TRANSACTION statement, you lock only those database resources necessary to complete each task. You can do this by using the RESERVING clause.

For example, you can name one relation (EMPLOYEES) from which you intend only to retrieve data, while naming other relations (COLLEGES and DEGREES) for update activities, as follows:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>      EMPLOYEES FOR SHARED READ,
cont>      COLLEGES FOR SHARED WRITE,
cont>      DEGREES FOR EXCLUSIVE WRITE
```

You can specify the following reserving options in the RESERVING clause of your START_TRANSACTION statement for update transactions:

- SHARED READ
- SHARED WRITE
- PROTECTED READ
- PROTECTED WRITE
- EXCLUSIVE READ
- EXCLUSIVE WRITE

Some database operations in a read/write transaction may require a higher level of record locking than the shared level. In such cases, Rdb/VMS automatically promotes locking to a protected read or protected write level to complete the task. Although the level of locking may often be higher than that which you specified, it is never lower than the level specified in the START_TRANSACTION statement.

You can access multiple databases in a single transaction and specify different transactions for relations in the different databases. The following example shows a read/write transaction that accesses relations in two databases; the relation (EMPLOYEES) in the first database is reserved for protected write access, and the relation (JOB_INFO) in the second database is reserved for shared read access.

```
RDO> INVOKE DATABASE DB1 = FILENAME 'PERSONAL$DISK:PERSONNEL'
RDO> INVOKE DATABASE DB2 = FILENAME 'PERSONAL$DISK:BENEFITS'
RDO> START_TRANSACTION ON DB1 USING
cont> (READ_WRITE RESERVING EMPLOYEES FOR PROT WRITE) AND
cont> ON DB2 USING (READ_WRITE RESERVING JOB_INFO FOR SHARED READ)
RDO>
```

If you omit the explicit reserving options, Rdb/VMS assumes the defaults.

The general form of the START_TRANSACTION RESERVING syntax is as follows:

```
START_TRANSACTION READ_WRITE
   RESERVING relation-name FOR  share-mode  lock-type
   WITH [NO]AUTO_LOCKING
```

Share mode can be one of the following:

- SHARED

  Other users can work with the same relation as you do. Depending on the option those users choose, they can have read-only, or read and write access to the relation.

- PROTECTED

  Other users can read records from the same relations as you, but cannot have write access.

- EXCLUSIVE

  Other users cannot even read records from your relation. If another user tries to access the same relation, Rdb/VMS denies the request.

Lock-type can be one of the following:

- READ

  You plan to retrieve records from relations without changing any of those records or storing new ones.

- WRITE

  You plan to retrieve and change records, or store new ones.

The WITH [NO]AUTO_LOCKING option is discussed in Section 2.3.4.1.

The effect of the reserving options you choose in your START_TRANSACTION statement depends on the options other users currently accessing the database have already specified. In an environment of multi-user database access, lock conflicts can cause delays or the need for special programming to handle the conflicts.

Note that a batch-update transaction works much like a read/write transaction with the EXCLUSIVE WRITE reserving option. However, unlike the EXCLUSIVE WRITE reserving option, a batch-update transaction locks the entire database rather than just specific relations; also, because there is no recovery-unit journal (RUJ) file with a batch-update transaction, you must be careful not to corrupt the database by issuing a ROLLBACK statement. A BATCH_UPDATE transaction is most useful for the initial loading of the database.

In all update cases, Rdb/VMS does not allow other transactions to read changed records until the updating transaction executes a COMMIT or ROLLBACK statement. Because Rdb/VMS locks your records against access by other users, you can display the changes you have made to those records. This record locking assures the consistency and integrity of database records.

The following sections discuss auto-locking and the specific share-mode and lock-type combinations; they are followed by a discussion of locking and lock conflict resolution.

2.3.4.1 Auto-Locking

Auto-locking is an option that causes tables referenced by constraints and triggers but not appearing in the RESERVING clause to be automatically locked when accessed from a constraint or trigger. By default, auto-locking is in effect when you specify the RESERVING clause; however, you can choose to disable it for the transaction by specifying NOAUTO_LOCKING.

One reason for the implementation of auto-locking is a problem that can arise when triggers (a feature available with Rdb/VMS V3.1) are defined for a database that had been used with Rdb/VMS V3.0 and has been converted for use with subsequent versions of Rdb/VMS. An application that ran under V3.0 is now run against a database that has triggers defined; however, without auto-locking, this application will fail because the relations specified in the trigger are not known to the START_TRANSACTION statement. Example 2–1 illustrates the behavior without and with auto-locking.

Example 2–1 Auto-Locking Versus No Auto-Locking

```
! Because of the trigger EMPLOYEE_ID_CASCADE_DELETE, a deletion from
! the EMPLOYEES relation will cause a "cascading deletion" of associated
! records in the DEGREES, JOB_HISTORY, and SALARY_HISTORY relation.
!
RDO> START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR
cont> EXCLUSIVE WRITE WITH NOAUTO_LOCKING
!
! The following DELETE statement fails because the tables in the
! triggered action have not been reserved and because you have specified
! no auto-locking.
!
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00165" ERASE E END_FOR
%RDB-E-UNRES_REL, relation DEGREES in specified request is not a relation
reserved in specified transaction
RDO> ROLLBACK
!
! Now, permit auto-locking (the default), and the subsequent ERASE
! statement is successful (including the triggered cascading deletions).
!
RDO> START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR
cont> EXCLUSIVE WRITE             ! Default = WITH AUTO_LOCKING
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00165" ERASE E END_FOR
RDO>
!
! Note, however, that you still cannot explicitly reference any of the
! other relations unless you explicitly include them in the RESERVING clause.
!
RDO> FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = "00165" PRINT JH.* END_FOR
%RDB-E-UNRES_REL, relation JOB_HISTORY in specified request is not a relation
reserved in specified transaction
RDO> ROLLBACK
```

Another problem resolved by auto-locking is the need (in versions of Rdb/VMS before 3.1) to change the RESERVING clauses of START_TRANSACTION statements after a constraint referring to another relation was added. With auto-locking, you no longer need to change the RESERVING clauses in such cases.

Note the following usage information about auto-locking:

- In a read-only transaction, any RESERVING clause can only declare READ locks, and auto-locking has no effect.
- In a read-write transaction with auto-locking, Rdb/VMS determines the lock specification for each table accessed by a constraint or trigger when the table is first accessed with a data manipulation statement from a constraint or trigger.
- If auto-locking is in effect and any of the tables referenced in a trigger or constraint definition also appears on the list of explicitly reserved tables, the explicitly specified lock mode must not conflict with the lock mode required by the constraint or trigger that references the table.

Note *SQL always uses auto-locking. There is no way to specify no auto-locking in SQL.*

2.3.4.2 Shared Read Reserving Option The shared read option lets other users' transactions retrieve records from the same relation you have accessed. It also allows transactions to update records within the same relation, except if those transactions are in an exclusive share mode. However, as you retrieve individual records from the relation, those individual records become unavailable for update by other users until you terminate your transaction.

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

2.3.4.3 Shared Write Reserving Option The shared write option lets other transactions retrieve or update records in the same relation you have accessed, but not the particular records you have locked. Updated versions of records from other transactions are not available to you until both your transaction and the other (updating) transactions terminate (with either a COMMIT or ROLLBACK statement). Also, any updated versions of the records you change are not available to other users until you terminate your update transaction with a COMMIT or ROLLBACK statement and other users begin new transactions.

Because many users can access the same relation, many records may be locked. Such record locking can result in access conflicts that can affect the performance and the level of concurrent access to database resources. Only one transaction can update any given record at one time. If another user has locked a record for update or has placed a write lock on a record for retrieval, you cannot access that record for update until the record is released by the locking transaction.

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

2.3.4.4 Protected Read Reserving Option The protected read option lets you read records from the same relation that other transactions are accessing. However, this option ensures that no other users can write to the relation that your transaction reserves in this manner. For example, assume you retrieve a record to generate a report, and that the contents of the record must be kept stable until the transaction is completed. The protected read option, unlike the shared read option, prevents other users from changing any records included in the report until the report is finished.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>     EMPLOYEES FOR PROTECTED READ
```

2.3.4.5 Protected Write Reserving Option The protected write option lets your transaction update the relation but prevents other transactions from updating that relation. Other users can only retrieve data from the relation. Therefore, a transaction with extensive updates may execute faster using the protected write option as opposed to the shared write option, because Rdb/VMS does not have to check for as many conflicting locks as in the shared write reserving option. Wherever possible, use indexed fields in your RSE so that only those database resources required by your transaction are locked; otherwise, Rdb/VMS will lock the entire relation.

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

2.3.4.6 Exclusive Read Reserving Option The exclusive read option allows only your transaction to read the specified relation; other users cannot read or update this relation. This reserving option uses the fewest locks because Rdb/VMS locks the resource at the relation level. Because there is no conflict with other users, the exclusive share mode retrieves data faster than shared or protected share modes. Specify EXCLUSIVE READ or PROTECTED READ in the START_TRANSACTION statement if you require a current picture of the database for retrieval, thereby ensuring that no other transaction that might change the records can start.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>     EMPLOYEES FOR EXCLUSIVE READ
```

2.3.4.7 Exclusive Write Reserving Option The exclusive write option lets only your transaction have access to the relation to read or update a record; other transactions are prevented from reading or updating any record in the relation. If you are doing updates to one or more relations and the transaction is fairly short, consider specifying the EXCLUSIVE WRITE option in your START_TRANSACTION statement. For example:

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

The exclusive write option is also the preferred method for loading data into databases in most circumstances, as opposed to using a batch-update transaction. (For a discussion of the batch-update transaction, including special notes and restrictions, see Section 2.3.3.)

When your transaction includes a MODIFY or ERASE statement, Rdb/VMS checks to see if another user has a lock on the record or records you need. If the record has no lock, Rdb/VMS locks it by putting an exclusive share mode lock on the record and executes the update statement. Your transaction holds the lock on this record until you commit the change to the database with a COMMIT statement or undo the change with a ROLLBACK statement.

2.3.5 Locking and Lock Conflict Resolution

Once Rdb/VMS grants the reserving option or options specified in your START_ TRANSACTION statement, it locks records identified by the RSE according to the kind of task, or verb, your transaction executes. For example, when you retrieve a record to display the values for certain fields, Rdb/VMS places read locks on them. However, when you issue a MODIFY statement, Rdb/VMS places a more restrictive write lock on the record, or records, so that no other transaction may intervene and change the values you intend to change.

When you lock a record for a read or write operation, you affect other users. Figure 2–1 shows what happens when you start a transaction reserving a relation for a specified type of access, and another user starts a transaction, attempting to reserve that relation for a specified type of access. (This table assumes that the WAIT option is specified.) If the other user's access causes a conflict, that user must either wait for record locks to be released (when WAIT is specified) or must terminate the active transactions and begin again.

Figure 2–1 Database Access Conflicts

If You Access a Record Using Transaction Mode:	Someone Else Using: READ_ONLY Has:	READ_WRITE SHARED READ Has:	READ_WRITE SHARED WRITE Has:	READ_WRITE PROTECTED READ Has:	READ_WRITE PROTECTED WRITE Has: [1]	READ_WRITE EXCLUSIVE READ Has:	READ_WRITE EXCLUSIVE WRITE Has:
READ_ONLY	No conflict	No conflict	No conflict	No conflict	No conflict [2]	A wait	A wait
READ_WRITE SHARED READ	No conflict	No conflict	No conflict to read, a wait to update	No conflict	No conflict to read, a wait to update [3]	A wait	A wait
READ_WRITE SHARED WRITE [2]	No conflict	No conflict	No conflict to read a record not updated; otherwise, a wait [1]	A wait	A wait	A wait	A wait
READ_WRITE PROTECTED READ	No conflict	No conflict	A wait	No conflict	A wait	A wait	A wait
READ_WRITE PROTECTED WRITE [2]	No conflict	No conflict	A wait	A wait	A wait	A wait	A wait
READ_WRITE EXCLUSIVE READ	A wait	A wait	A wait	A wait	A wait	A wait	A wait
READ_WRITE EXCLUSIVE WRITE [3]	A wait, then a conflict	A wait	A wait	A wait	A wait	A wait	A wait

[1] If index is used. If index is not used, may lock entire relation.
[2] Updates are written to SNP file.
[3] Updates are not written to SNP file.

ZK−1483A−GE

2.3.6 Other START_TRANSACTION Options

In addition to the reserving options of the START_TRANSACTION statement described earlier, you can specify other options that affect how Rdb/VMS handles your transactions:

- Constraints
  - Evaluating at verb time
  - Evaluating at commit time
- Options
  - Wait
  - Nowait
  - Consistency
  - Concurrency

The following sections discuss these options.

2.3.6.1 Evaluating Constraints at Verb Time You can define constraints to check for values you want to store in the database. By default, Rdb/VMS evaluates each user-defined constraint at the time specified in the constraint definition with the CHECK ON clause (default is CHECK ON UPDATE). However, you can override the CHECK ON clause by specifying the EVALUATING clause in the START_TRANSACTION statement.

You can specify that Rdb/VMS should evaluate the constraints at verb time (VERB_TIME) or commit time (COMMIT_TIME). By specifying VERB_TIME, you indicate that Rdb/VMS should evaluate the constraint when the statement to store, modify, or delete data executes. (Evaluation at commit time is discussed in Section 2.3.6.2.)

Evaluating constraints at verb time can make it easier to isolate which record is violating a constraint. Each time Rdb/VMS executes a STORE or MODIFY statement, the record stream your RSE identifies may contain one record or many records. When the record stream contains only one record, and an error occurs, you can handle that error by displaying the offending record or writing it to an exception file. On the other hand, if the record stream identifies more than one record in a FOR . . . END_FOR block that contains a STORE or MODIFY statement, and an error occurs, you want to be sure which record in the record stream has violated the constraint definition. So, for each execution of the STORE or MODIFY statement in the FOR . . . END_FOR block, you can specify that Rdb/VMS check the constraint by including the EVALUATING AT VERB_TIME clause.

Additionally, when you include update tasks in a host language program, you can handle errors with the ON ERROR clause. Specifying constraint evaluation at verb time causes control to pass immediately to the error handling statements. If your transaction waits until commit time to evaluate the constraint, Rdb/VMS may not signal the error at the verb level because the STORE or MODIFY statement will have completed. Refer to Chapter 10 for more details on error trapping and error handling using constraints.

If your transactions contain several update operations using both STORE and MODIFY statements, you may need to evaluate constraints at verb time to detect which operation or constraint caused the violation. Evaluating constraints at commit time may direct the entire transaction to roll back if error handling is not included at the verb level.

2.3.6.2 Evaluating Constraints at Commit Time If you have access to all referenced relations, you can evaluate constraints at commit time. You can defer constraint evaluation until you are ready to terminate your transaction. Rdb/VMS then checks each value against the defined constraint before allowing the record to be stored.

The main benefits of evaluating at commit time as opposed to verb time are as follows:

- You can store or modify records that depend on other records.

  If a field has a constraint requiring the existence of another record with a matching field value, then evaluating the constraint at commit time allows you to change the field value, make any other necessary changes, and then commit those changes, at which time the existence-checking constraint is evaluated. For example, assume that a database permitted department codes to be changed, and assume that a constraint required each employee's assigned department code to match an existing department code. If you evaluated constraints at verb time, changing the department code would cause an immediate violation; however, if you evaluated at commit time, you could change the department code, then change the code in the affected employee records, and then commit the changes.

- You can improve application performance.

  When you specify constraints to be evaluated at commit time, you defer the expense of evaluation (that is, the time required) until you enter the COMMIT statement. For example, assume you need to modify most of the records in a specific relation. You can start the transaction with the exclusive write reserving option to avoid access conflicts with other users and to reduce the use of lock resources, thus allowing your task to complete more efficiently. If at commit time you find you have numerous constraint violations, you can roll back the transaction, correct the erroneous values, and try the update operation again.

If your tasks include numerous changes to the database in a scheduled production update run, there may be very little conflict with other users accessing the database. In such cases, you can experiment with both VERB_ TIME and COMMIT_TIME constraints to see which meets your needs. You can enhance performance by ensuring that fields used in the constraint definition are indexed fields. Indexes allow Rdb/VMS to locate specific records efficiently. See the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for information on enhanced performance with indexed fields.

The following example shows an EVALUATING clause in a START_ TRANSACTION statement overriding the CHECK ON clause in the DEFINE CONSTRAINT statement. The DEFINE CONSTRAINT statement specifies a constraint called SH_EMP_ID_EXISTS and specifies that this constraint is to be evaluated for any new data stored in the database:

```
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 UPDATE.
```

The START_TRANSACTION statement overrides the CHECK ON UPDATE clause by deferring evaluation to commit time. (The constraint verifies that an EMPLOYEE_ID value exists in the EMPLOYEES relation before a SALARY_ HISTORY record can be stored.)

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>     EMPLOYEES FOR EXCLUSIVE WRITE,
cont>     SALARY_HISTORY FOR EXCLUSIVE WRITE EVALUATING
cont>     SH_EMP_ID_EXISTS AT COMMIT_TIME
```

2.3.6.3 Wait and Nowait Options You can specify how Rdb/VMS is to handle your transactions when you attempt to retrieve or update a resource (record, relation, or index) locked by another user. For example, you can elect to wait for locked records to be released by specifying WAIT (the default) in your START_TRANSACTION statement; or you can specify NOWAIT, in which case Rdb/VMS returns an error message that a record is unavailable, and you can then terminate the current transaction and enter your START_TRANSACTION statement again or start another transaction. The following example specifies the NOWAIT option:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>     EMPLOYEES FOR PROTECTED WRITE,
cont>     JOB_HISTORY FOR PROTECTED WRITE,
cont>     SALARY_HISTORY FOR SHARED READ NOWAIT
```

The nowait option can be used to program flexibility into an application, such as when you might wish to allow the user to choose whether or not to wait, or to allow a certain number of retries before informing the user that the record is unavailable. For example, to permit the user to choose whether or not to wait, your program might contain the following logic.

1 Start a transaction specifying NOWAIT.

2 Start the data manipulation operation.

3 On error (that is, lock conflict), display a message to the user: "The record you want is in use. Do you wish to wait?"

4 If the user replies Yes, start a transaction specifying WAIT. (If the user replies No, handle the condition as appropriate for the application.)

If you specify or accept the default of WAIT in your START_TRANSACTION statement, you should consider the possibility of encountering incompatible transaction modes. For example, consider the following sequence of events, where User B must wait until User A completes a transaction. User A starts a transaction and reserves a relation specifying EXCLUSIVE WRITE access. This means that any other transactions cannot gain snapshot access to the database resources held by the exclusive lock until the transaction specifying EXCLUSIVE WRITE is terminated. User B's snapshot request, therefore, is not compatible with an EXCLUSIVE WRITE lock, and so Rdb/VMS immediately causes User B to wait.

1 User A starts a read/write transaction, reserving a relation for exclusive write access, and fetches a collection of records.

2 User B then starts a read-only transaction accessing the same relation and includes the WAIT clause.

3 When User B attempts to access the records in a FOR . . . END_FOR block to display certain values, Rdb/VMS causes User B to wait.

4 When User A completes its transaction with a COMMIT or ROLLBACK statement, Rdb/VMS then returns the "lock conflict" error to User B and permits User B to resume processing.

A read/write transaction that specifies EXCLUSIVE WRITE reserving option does not write data to the snapshot file; it is *always* incompatible with access requests from read-only transactions. Rdb/VMS defines read-only transactions in such a way that all data committed to the database before the start of the transaction must be available to the transaction that requests access to the snapshot file. Because it is impossible for Rdb/VMS to determine whether or not the transaction using the exclusive write reserving option may have written data to the database, it cannot satisfy the read-only transaction's requirements.

2.3.6.4 Consistency and Concurrency Options

The consistency and concurrency options are provided for compatibility with other Digital relational database products, such as VAX Rdb/ELN. In Rdb/VMS, the distinction is not meaningful. If you specify CONCURRENCY, Rdb/VMS translates that to CONSISTENCY. Rdb/VMS always guarantees degree 3 consistency. Degree 3 consistency means that the database system guarantees that data you have read will not be changed by another user before you issue a COMMIT statement.

In other relational systems that you might access using the remote feature of Rdb/VMS, the consistency option specifies the degree to which you want to control the consistency of the database. In such systems, the concurrency option sacrifices some consistency protection for improved performance with many users.

2.3.7 Indexes

Rdb/VMS can use indexes to locate specific records using the database key for those records. A database key, or **dbkey**, is a pointer or address that indicates a specific record in the database. There is a separate index (B-tree) structure for each sorted index defined (by the user or by Rdb/VMS) in the database. Each B-tree structure is created by linking index nodes together in a balanced hierarchical structure. These nodes are linked consecutively according to the index definition. By default, these nodes are in ascending sequence and are horizontally linked in low-to-high key value. If you define a descending index, the index nodes are horizontally linked in high-to-low key value. The links between the nodes are created by using the dbkeys. Thus, updating the index fields of records means updating index nodes as well.

During the database design phase, the database administrator or owner of the database should identify certain fields in each relation as primary keys and foreign keys. Primary and foreign key fields are usually indexed.

A **primary key** is the field (or group of fields) that you select to be the principal identifier of each record in a relation. It is best if the field you select as a primary key is unique and stable, because the number of input/output operations necessary to update an index is high, and because the likelihood of locking contention increases. Therefore, you can use the primary key to locate a specific record, and update other, non-indexed fields, in those records. In this way, you benefit from the efficient access methods Rdb/VMS uses to locate the records you need, but you do not suffer the overhead penalty of updating the index nodes. A **foreign key** is a field or group of fields in one relation that has a matching value in the primary key of another relation. You can use foreign keys for joins, regardless of whether an index is defined for the foreign key.

You should decide which fields are important to index, to reduce the number of write locks on the records in the relation and thereby reduce the chances of processing delays and potential deadlocks. For example, you can start a transaction specifying the SHARED WRITE reserving option. Another user can enter an identical START_TRANSACTION statement to read or update records in the same relation you have accessed. If no indexes are defined for the key field, Rdb/VMS must physically scan each record in the database itself, placing a write lock on the entire protected relation. The other transaction attempts to select records from the same relation and conflicts with your transaction because your transaction has already placed locks on those records. Your transaction may even promote the locking to the exclusive level, and allow no other user to access any of the records in the relation. Other users must terminate their transactions and enter the START_TRANSACTION statement again to select the records or wait until the records in that relation are available.

On the other hand, if the fields you use to select records for your transaction are indexed, Rdb/VMS can refer to the index tables to locate only the records you need. Rdb/VMS will also place read locks only on the index nodes that contain the dbkeys to those records and thus allow other users to access the remaining records in the relation. Thus, you should use indexes to locate records, and to increase database concurrency by reducing possible deadlocks and by making more resources available to other database operations. For further information on primary and foreign keys and indexes, see the *VAX Rdb/VMS Guide to Database Maintenance and Performance*.

2.3.8 Transaction Scope

Remember, a transaction is a unit of database activity you perform with one statement or many statements. The START_TRANSACTION statement that marks the start of a transaction and the COMMIT or ROLLBACK statement that terminates the transaction identify the **scope** of the transaction. Rdb/VMS executes either all of the statements in the scope of the transaction or none of them. Before you begin your transaction, you should determine the tasks you want to accomplish. Some of these tasks might be:

- Data retrievals from the database
- Changes you want to make to existing records in the database
- Changes to the data definitions

It is usually wise to limit the scope of a transaction to a particular type of task. If you mix tasks in a transaction, you may want to undo some tasks and keep others. By restricting each transaction to a specific task, you can roll back certain operations and make others permanent. Moreover, in certain applications, attempting to do too much in a single transaction can cause the process to exceed its quotas for locks or virtual memory.

It is recommended that you perform any terminal input/output operations outside the scope of a read/write transaction. Instead, use the following approach:

1 Gather data from the terminal.

2 Perform the transaction.

3 Write the results to the terminal.

2.3.9 Ending a Transaction

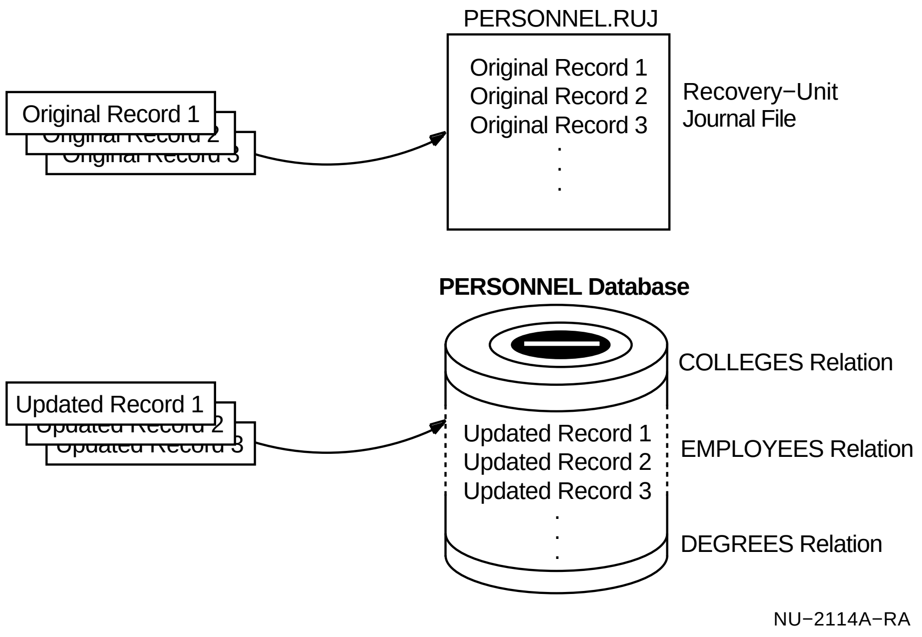
All transactions end normally with a COMMIT or a ROLLBACK statement. This section discusses the effect of each statement, including the impact on the RUJ file.

An update transaction can physically change the values in the database. In the following example, the PERSONNEL file is invoked and a START_TRANSACTION statement reserves the EMPLOYEES relation for shared write access for an update transaction:

```
RDO> INVOKE DATABASE FILENAME PERSONNEL
RDO> START_TRANSACTION READ_WRITE
cont>    RESERVING EMPLOYEES FOR SHARED WRITE
```

Before each update is physically written to the database, the original record is written to the recovery-unit journal file (file type RUJ). Each user who performs an update has an RUJ file in his or her SYS$LOGIN directory (or other location specified by the logical name RDMS$RUJ) for the life of a transaction. After all the updated records have been written to the database, the EMPLOYEES relation has new records added to it. Figure 2–2 shows the effect of an update on a database.

Figure 2–2 Recovery-Unit Journal File During an Update Transaction



You can terminate an Rdb/VMS transaction with either of the following statements:

- COMMIT

  Use the COMMIT statement to make your changes permanent. This causes Rdb/VMS to invalidate the RUJ file and to make it ready for further transactions.

- ROLLBACK

  Use the ROLLBACK statement to undo the changes you have made to the database within the scope of a transaction. The ROLLBACK statement uses the RUJ file to bring the database back to its pretransaction state.

Figure 2–3 shows the effect of a COMMIT statement on a database.

Figure 2–3 Recovery-Unit Journal File with COMMIT

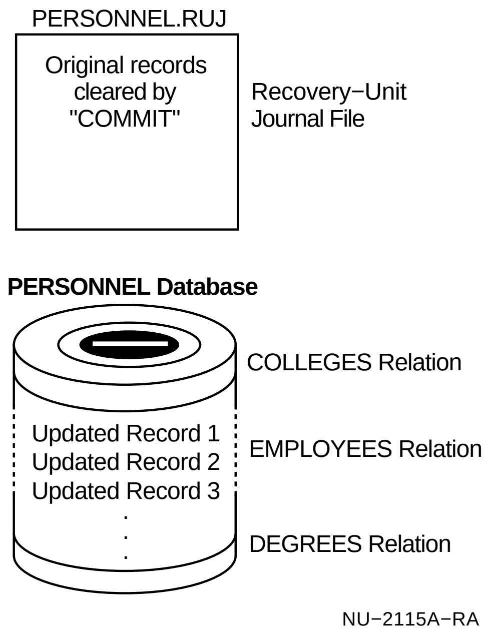
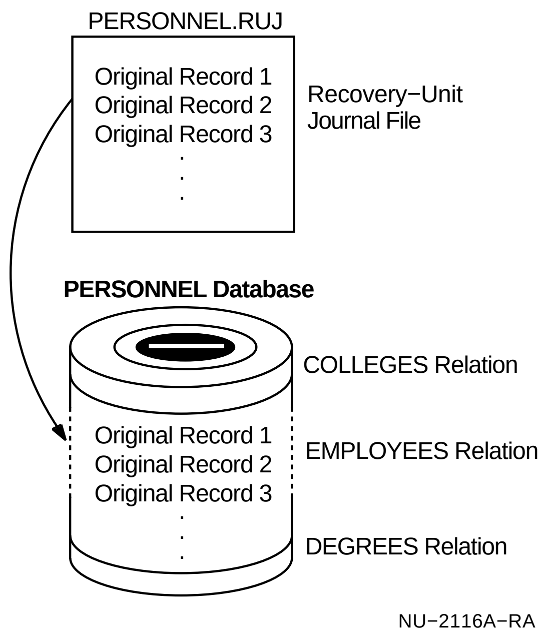


Figure 2–4 shows the effect a ROLLBACK statement has on the database.

Figure 2–4 Recovery-Unit Journal File with ROLLBACK



Because the updates actually change the state of the database, the RUJ file is used to return the database to its pretransaction state by writing the original records back to the database. When the transaction terminates, the EMPLOYEES relation is unchanged.

2.4 The Query Optimizer

Because a relational database model represents the user's view of the data stored in the database, determining the best way to retrieve that data can be a very complex task. Rdb/VMS contains a query optimizer that automatically analyzes each query to determine the most efficient method of access to the data. Efficiency can be measured as the number of disk accesses required to retrieve data values in the database.

The query optimizer is a sophisticated component of Rdb/VMS that uses a combination of algorithms to evaluate the query and arrive at a low-cost solution to retrieve the data in the database. The order in which you specify joins and the order of the clauses in the RSE do not, in most cases, influence the order the query optimizer uses to satisfy your query. You can, however, help the query optimizer by defining indexes for fields you use frequently in your queries.

To evaluate a query, the query optimizer:

- Develops alternative solutions for retrieving data
- Associates a cost factor with each solution based on estimated input/output requirements
- Chooses the most cost-effective solution in terms of the least number of estimated input/output operations required to fetch the record

The query optimizer evaluates every query in terms of an estimate of efficient access, so you do not have to be overly concerned about how to construct your queries. As a database designer, however, you can assist the query optimizer by extending its access options. For example, if your query includes only those fields for which indexes are defined, you are providing the query optimizer with an option to retrieve data directly from the index without scanning the relation sequentially.

The query optimizer may use only the index if it contains all the data necessary to satisfy the query, or if the index provides a useful ordering of records. For example, when a query names one field in the RSE and two other fields in the print list, all three fields must have indexes defined for them in order for the query optimizer to choose an access method that uses only the index. On the other hand, the algorithms the query optimizer uses may result in its not using the index on that field at all, if the query optimizer estimates that the data can be retrieved directly from the relation with fewer input/output operations than by using the index or indexes.

The following list describes some of the tasks the query optimizer performs to find the best solution for a query that contains one or more CROSS clauses. The query optimizer:

- Breaks down a query into equivalent sequences of two relational joins
- Finds the best way to perform each join based on the estimated relative cost of each access method
- Estimates cardinality (number of records to be retrieved) of each join based on a "join predicate," (the RSE supplied by the user), and the presence of indexes for specific fields
- Determines overall cost of each strategy
- Selects minimum cost strategy

The query optimizer chooses one of the following methods for retrieving data from a relation:

- Sequential retrieval

  Accesses the database pages for a relation sequentially and searches for the field values of the records directly on the pages.

- Index retrieval

  Accesses one or more index structures and retrieves the dbkey of a record. Rdb/VMS then uses the dbkey to directly access the data record to which it belongs.

- Index-only retrieval

  Accesses only the index data. If the desired data is located in an index key, Rdb/VMS can obtain the data without going to the actual data record.

- Dbkey retrieval

  Accesses the relation's data directly through the dbkey (logical location) record pointer.

2.5 Sample Interactive Session Using the START_TRANSACTION Statement

The following interactive session shows the read-only and read/write versions of the basic START_TRANSACTION statement:

```
! The statements in the scope of the first transaction
! merely examine the database. The transaction does not
! change any values.
!
RDO> START_TRANSACTION READ_ONLY

!
! Display the number of records in
! the EMPLOYEES relation.
!
RDO> PRINT COUNT OF E IN EMPLOYEES
 101

!
! How many employees live in Rochester?
!
RDO> PRINT COUNT OF E IN EMPLOYEES WITH E.CITY = "Rochester"
 7
```

```
!
! If you attempt to change the database by erasing all
! Rochester records with the ERASE statement:
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   ERASE E
cont> END_FOR
!
! You are attempting to update the database. RDO returns an
! error message:
!
%RDB-F-READ_ONLYTRANS, attempt to update from a READ_ONLY transaction

!
! Display information from the records
! of employees who live in Rochester.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  PRINT
cont>   E.LAST_NAME,
cont>   E.FIRST_NAME,
cont>   E.EMPLOYEE_ID,
cont>   E.CITY
cont> END_FOR
 LAST_NAME     FIRST_NAME   EMPLOYEE_ID  CITY
 Vormelker     Daniel       00242        Rochester
 Edwards       Keith        00254        Rochester
 Orlando       Johanna      00269        Rochester
 DuBois        Alvin        00275        Rochester
 Chase         Stan         00336        Rochester
 Boudreau      Wes          00346        Rochester
 Stornelli     Franklin     00437        Rochester

!
! Terminate the read-only transaction scope with a
! COMMIT or ROLLBACK statement.
!
RDO> COMMIT

!
! Now start a new transaction that allows changes to be
! written to the database.
!
RDO> START_TRANSACTION READ_WRITE

!
! Display information from the records
! of employees who live in Rochester.
!
```

```
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  PRINT
cont>   E.EMPLOYEE_ID,
cont>   E.CITY,
cont>   E.POSTAL_CODE
cont> END_FOR
 EMPLOYEE_ID   CITY        POSTAL_CODE
 00242         Rochester   03867
 00269         Rochester   03867
 00275         Rochester   03867
 00336         Rochester   03867
 00346         Rochester   03867
 00437         Rochester   03867

!
! Change the value of the POSTAL_CODE field for all the
! Rochester records.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  MODIFY E USING
cont>   E.POSTAL_CODE = "03801"
cont>  END_MODIFY
cont> END_FOR

!
! Verify the change.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  PRINT
cont>   E.EMPLOYEE_ID,
cont>   E.CITY,
cont>   E.POSTAL_CODE
cont> END_FOR
 EMPLOYEE_ID   CITY        POSTAL_CODE
 00242         Rochester   03801
 00269         Rochester   03801
 00275         Rochester   03801
 00336         Rochester   03801
 00346         Rochester   03801
 00437         Rochester   03801


!
! Delete (erase) the Rochester records.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  ERASE E
cont> END_FOR

!
! Are there any Rochester records remaining?
!
RDO> PRINT COUNT OF E IN EMPLOYEES WITH E.CITY = "Rochester"
 0
```

```
!
! Check to see that records are deleted.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>  PRINT
cont>   E.*
cont> END_FOR
! (No records are displayed)

!
! Add a new record.
!
RDO> STORE E IN EMPLOYEES
cont>  USING
cont>   E.EMPLOYEE_ID = "00502";
cont>   E.LAST_NAME = "Towne";
cont>   E.CITY = "Manchester";
cont>   E.POSTAL_CODE = "03103"
cont> END_STORE

!
! Verify the addition of the record.
!
RDO> FOR E IN EMPLOYEES WITH E.LAST_NAME = "Towne"
cont>  PRINT
cont>   E.EMPLOYEE_ID,
cont>   E.LAST_NAME,
cont>   E.CITY,
cont>   E.POSTAL_CODE
cont> END_FOR
 00502  Towne      Manchester       03103

!
! If you want to make the changes to the database permanent, enter
! the COMMIT statement.
!
! If you do not want the changes applied to the database,
! enter the ROLLBACK statement. If you enter ROLLBACK, the
! Rochester records are retained in the database with no
! changes and the Manchester record of the employee named
! Towne is not added.
!
RDO> COMMIT
RDO> FINISH    !(optional)
RDO> EXIT
```

3 Using Record Selection Expressions

This chapter shows you how to use record selection expressions (RSEs) to select and display values from a database. You use an RSE to select a group of records and then to manipulate the data from those records. Note that Rdb/VMS lets you include database queries that use either embedded data manipulation statements or Callable RDO in your application programs. (Callable RDO is discussed in detail in Chapter 19.)

3.1 Forming Streams of Records

A record stream can consist of all or only some of the records in a relation. You can form a record stream using any of the following:

- A FOR statement
- A DECLARE_STREAM or START_STREAM statement

In these statements, an RSE identifies the records that form the record stream. For further information on specifying and using record streams, see Section 6.4.

Having chosen the records you wish to retrieve, you enter a PRINT statement in RDO to specify what fields you want displayed. The PRINT statement displays data on the terminal so you can be certain you have selected the correct records.

Once you have tested your query and want to include it in a program, this display feature is no longer necessary. Later chapters in this manual provide details about converting your RDO queries to host language application programs.

In some cases, you may have been denied access to an entire relation or certain fields within a relation. If you have been denied access to a field or a relation, you will get the following error when you try to form your record stream:

```
%RDB-E-NO_PRIV, privilege denied by database facility
```

3.2 Retrieving All the Records in a Relation

One of the simplest operations in Rdb/VMS is selecting all the records in a relation.

The following FOR statement contains an RSE that forms a record stream consisting of all the records in the EMPLOYEES relation:

```
FOR E IN EMPLOYEES
```

The expression E IN EMPLOYEES is an RSE that selects records from the EMPLOYEES relation. This RSE includes every record of the EMPLOYEES relation in the record stream.

The character E in the first line of the RSE is a context variable. A **context variable** is a temporary name you assign to the record stream created by the RSE. In subsequent lines of the RSE, the context variable and a period (.) appear before each field name. By qualifying each field name with the context variable and a period, you indicate clearly to RDO the relation to which each field belongs. If an RSE statement refers to more than one relation, assign a unique context variable to each relation. (Context variables can be up to 31 characters long; however, try to choose a context variable that you can easily associate with the relation, such as the first letter of the relation name or some other meaningful abbreviation.)

The RDO block, FOR . . . END_FOR, includes the RSE and identifies a record stream. Other statements included in the FOR . . . END_FOR block operate on each record in this record stream. For example, to display data about all employees in the EMPLOYEES relation, you use the PRINT statement. The complete query is shown in the example that follows:

```
FOR E IN EMPLOYEES
 PRINT
  E.LAST_NAME,
  E.FIRST_NAME,
  E.EMPLOYEE_ID
END_FOR
```

The preceding example displays three fields from each record in the EMPLOYEES relation. Note the following rules about using the RDO PRINT statement:

- Qualify each of the field names with the context variable associated with the field's relation.

- Use commas to separate the expressions in the list.

The RSE selects records for inclusion in the record stream. The PRINT statement retrieves one record at a time and specifies fields from those records to be displayed. You can also display data for all the fields in the relation by using a special format of the PRINT statement. Instead of specifying each field name individually, substitute an asterisk (*) for the list of field names. For example:

```
FOR E IN EMPLOYEES
  PRINT E.*
END_FOR
```

RDO prints each of the field names in the relation as the first line of the display. When a relation such as EMPLOYEES contains many fields, RDO wraps the remainder of a long record onto the next line of your terminal. The resulting display can be difficult to read.

For more readable and flexible displaying of data from a database, it is suggested that you use a product designed for data selection and formatting, such as DATATRIEVE, RALLY, DECdecision, or TEAMDATA. However, you can still create a somewhat readable display of records with many field values by using RDO. Create a command file with the file type RDO (for example, REPORT1.RDO). In this file, you can include a query such as the one in the preceding example, and specify an output file with the SET OUTPUT statement. (Do not use both the SET VERIFY and SET OUTPUT statements; the SET VERIFY statement will cause each statement you enter to appear twice in the output file.) To close the output log file, type SET NOOUTPUT, or type SET OUTPUT without specifying a destination file name.

The file REPORT1.RDO might contain the following statements:

```
SET NOVERIFY
SET OUTPUT REPORT1.LOG
FOR E IN EMPLOYEES
  PRINT E.*
END_FOR
SET NOOUTPUT
```

You can execute this command file by typing an at sign (@), followed by the name of the command file (the default file type is RDO):

```
RDO> @REPORT1
```

You can then print the file, REPORT1.LOG, that contains the results of a command file, REPORT1.RDO, using the wide-line printer format instead of the 80-character limit of an interactive terminal; or you can use the DCL command SET TERMINAL/WIDTH=132 and display the file on your screen.

You can direct RDO to display special character strings or literal expressions by using the PRINT statement and quotation marks to enclose the string. You can combine literals with value expressions such as the statistical expression COUNT, separating each element with a comma. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for a discussion of statistical expressions. The following query displays literal expressions and value expressions:

```
RDO> PRINT "Number of records in EMPLOYEES = ", COUNT OF E IN EMPLOYEES
Number of records in EMPLOYEES = 101
```

The following example shows you how to use a command file to format and display a simple report that shows the number of records in each relation of the sample personnel database:

```
SET NOVERIFY
SET OUTPUT COUNT.LOG
PRINT "  "
PRINT "Statistics for database PERSONNEL follow: "
PRINT " "
PRINT "Count of Employees -------> ", COUNT OF E IN EMPLOYEES
PRINT "Count of Jobs ------------> ", COUNT OF J IN JOBS
PRINT "Count of Degrees ---------> ", COUNT OF D IN DEGREES
PRINT "Count of Salary_History --> ", COUNT OF SH IN SALARY_HISTORY
PRINT "Count of Job_History -----> ", COUNT OF JH IN JOB_HISTORY
PRINT "Count of Work_Status -----> ", COUNT OF W IN WORK_STATUS
PRINT "Count of Departments -----> ", COUNT OF D IN DEPARTMENTS
PRINT "Count of Colleges --------> ", COUNT OF C IN COLLEGES
PRINT " "
PRINT "Statistics Complete for Database: PERSONNEL"
PRINT "  "
```

3.3 Displaying Records in Sorted Order

Use the SORTED BY clause of the RSE to signal RDO to order the records in a record stream. The default is *ascending* order.

The following example arranges EMPLOYEES records in alphabetical order by state:

```
FOR E IN EMPLOYEES SORTED BY E.STATE
  PRINT
    E.STATE,
    E.CITY,
    E.EMPLOYEE_ID
END_FOR
```

A field name on which the sort order of records is based is called a **sort key**. In the preceding example, there is one sort key (the STATE field). Because the query in the preceding example has only one sort key, it does not specify how RDO should arrange two or more records that have the same value for STATE. If you want to include cities in alphabetical order (A to Z) within the same state, use two sort keys, STATE and CITY, as in the following example:

```
FOR E IN EMPLOYEES SORTED BY E.STATE, E.CITY
  PRINT
    E.STATE,
    E.CITY,
    E.EMPLOYEE_ID
END_FOR
```

Specifying the CITY field as a second sort key ensures that Rdb/VMS arranges records with different values for the CITY field alphabetically within the same state. When you use more than one sort key, the first key is the **major sort key** and all other keys are **minor sort keys**. In the preceding example, the STATE field is the major sort key and the CITY field is a minor sort key.

Note *Rdb/VMS does not guarantee the order of the records retrieved unless you specify a sort key. You cannot assume that Rdb/VMS arranges records according to the values for any index key field of the relation. To control the arrangement of the records that Rdb/VMS displays, specify one or more sort keys.*

3.3.1 Indicating Ascending or Descending Sort Order

When you use a sort key, RDO normally arranges the records in ascending order by that key according to the standard ASCII collating sequence; that is, Rdb/VMS arranges numeric values in numerical order and character fields in alphabetical order. To reverse the order, use the keyword DESCENDING. [1]

You can sort a record stream by ascending values for one field and descending values for another field. To arrange the records in alphabetical order for the major sort key (STATE) and in reverse alphabetical order for the minor sort key (CITY), specify an explicit order for each field, as in the following example:

```
FOR E IN EMPLOYEES
 SORTED BY ASCENDING E.STATE,
      DESCENDING E.CITY
      PRINT
       E.STATE,
       E.CITY,
       E.EMPLOYEE_ID
 END_FOR
```

Now the records are in alphabetical order according to the major sort key, STATE, and within each state's grouping the records are in descending alphabetical order (Z to A) according to the minor sort key, CITY.

[1] Defining an index on the sort key field or fields can improve performance if the sort order and the type of index (ascending or descending) are the same. If the sort key does not have an index defined, or if the index cannot be used in the sort, Rdb/VMS must use the VMS Sort utility, thus causing a possible decrease in sort speed.

Unless you explicitly specify the sort order for each minor sort key, the default sort order of any minor key is the same as the order for the last explicit or default sort key. For example, if the major key is in ascending sequence (by default or explicitly specified) and the first minor key has descending sequence specified, the default sort order for any other minor keys is descending (unless and until a subsequent minor key has ascending sequence specified).

In the following example, because the RSE specifies a descending sort order for the major sort key, E.CITY, RDO sorts the other two minor sort keys, E.STATE and E.POSTAL_CODE, in descending order also:

```
FOR E IN EMPLOYEES
 SORTED BY DESCENDING E.CITY, E.STATE, E.POSTAL_CODE
```

3.3.2 Using Value Expressions as Sort Keys

A sort key can also be a value expression that refers to one or more fields in a relation. A **value expression** is a symbol or string of symbols used to calculate a value. When you use a value expression in a statement, Rdb/VMS calculates the value associated with the expression and uses that value when executing the statement. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for more information on value expressions.

For example, assume that you wish to display information about job codes, starting with the job code with the smallest salary range (difference between maximum and minimum salary). You could specify a sort key consisting of a value expression that calculates the salary range for each job code. The following query sorts the records of the JOBS relation by range value, beginning with the smallest range:

```
FOR J IN JOBS SORTED BY
   (J.MAXIMUM_SALARY - J.MINIMUM_SALARY)
 PRINT
   (J.MAXIMUM_SALARY - J.MINIMUM_SALARY),
   J.JOB_CODE,
   J.MAXIMUM_SALARY,
   J.MINIMUM_SALARY
END_FOR
```

To reverse the order of displayed values, simply include the explicit sort qualifier DESCENDING:

```
FOR J IN JOBS SORTED BY
   DESCENDING (J.MAXIMUM_SALARY - J.MINIMUM_SALARY)
 PRINT
   (J.MAXIMUM_SALARY - J.MINIMUM_SALARY),
   J.JOB_CODE,
   J.MAXIMUM_SALARY,
   J.MINIMUM_SALARY
END_FOR
```

3.4 Restricting the Number of Records: The FIRST Clause

RDO lets you experiment with different queries to find the ones best suited to your programming needs. For example, you normally do not need to display all the records of the database to see whether your queries work correctly. RDO has a special clause, FIRST *n*, that limits the number of records you display. The integer (*n*) in the FIRST *n* clause tells RDO how many records to retrieve.

Assume you must display the first ten records from the EMPLOYEES relation, and need to look at only three fields, STATE, CITY, and EMPLOYEE_ID, from each record. The records retrieved by RDO are not in any specific order. As you update the contents of the EMPLOYEES relation by adding, erasing, or modifying records, the order of records stored in the database changes. Therefore, unless you specify to RDO the order in which you want the records displayed, the FIRST 10 clause retrieves what might appear to be 10 random records. See Section 3.3 for details on the SORTED BY clause.

```
FOR FIRST 10 E IN EMPLOYEES
 PRINT
   E.STATE,
   E.CITY,
   E.EMPLOYEE_ID
END_FOR
```

Note *If the RSE does not specify a sort order, you cannot predict which ten records RDO will display. When you use the SORTED BY clause in the RSE, the FIRST* n *clause takes the specified number of records from the* sorted *records. RDO does the sort first, then displays the number of records specified in the FIRST* n *clause from this sorted order. Remember to use a SORTED BY clause when you begin an RSE with a FIRST* n *clause.*

3.5 Specifying Conditions to Retrieve Records: Relational and Logical Operators

Assume you want to find all employees in the PERSONNEL database whose last name is Toliver.

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = "Toliver"
  PRINT
    E.FIRST_NAME,
    E.LAST_NAME,
    E.EMPLOYEE_ID
END_FOR
```

The clause, WITH E.LAST_NAME = “Toliver”, is a *conditional expression*. It is equivalent to:

```
If LAST_NAME = "Toliver"
```

The value of this conditional test for a record is either true or false, depending on whether the field value in that record satisfies the condition (true), or does not satisfy the condition (false). A **conditional expression** restricts the record stream to those records that satisfy the condition. If a conditional expression for a record is false, RDO will not include that record in the record stream.

The equal sign (=) is a **relational operator** because it links a data value of a field or other value expression to a value. The relational operator EQUAL (=) is case sensitive. This means that RDO reads "Toliver" and "toliver" as two different character strings. In this case, if you specify a value "toliver" for an employee record stored in the database as "Toliver", RDO does not find the record you want.

In a program, the following conditional expression tests the value of the database field named LAST_NAME (in the EMPLOYEES relation) and the value of a host language variable named LAST-NAME:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = LAST-NAME
```

Here, the value of the conditional expression depends on the current value of the host language variable LAST-NAME. Table 3–1 summarizes most of the relational operators.

Table 3–1 RDO Relational Operators

Permitted Symbols	Relational Operation
EQ =	True if the two value expressions are equal.
NE <>	True if the two value expressions are not equal.
GT >	True if the first value expression is greater than the second.
GE >=	True if the first value expression is greater than or equal to the second.
LT <	True if the first value expression is less than the second.
LE <=	True if the first value expression is less than or equal to the second.
BETWEEN	True if the first value expression is equal to or between the second and third value expressions.
ANY	True if the record stream specified by the RSE includes at least one record. If you add NOT, the condition is true if there are no records in the record stream.

(continued on next page)

Table 3–1 (Cont.) RDO Relational Operators

Permitted Symbols	Relational Operation
MATCHING	True if the second expression matches a substring of the first value expression. Not case sensitive. MATCHING uses these special characters: * Matches any string in that position % Matches any character in that position
MISSING	True if the value expression is null. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for information on missing values.
UNIQUE	True if the record stream specified by the RSE includes only one record. If you add NOT, the condition is true if there is more than one record in the record stream or if the record stream is empty.
CONTAINING	True if the string specified by the second string expression is found within the string specified by the first. Not case sensitive.
STARTING WITH	True if the first characters of the first string expression match the second string expression. Case sensitive.

Note *In all cases except the MISSING operator, if either value expression is null, the value of the condition is null.*

The collating sequence for a field determines the behavior of relational operators in comparisons of two fields or of a field with a literal value. If you use any collating sequence besides the standard ASCII sequence, note the following specific behaviors:

- CONTAINING relational operator

  This operator is not sensitive to diacritical markings nor is it case sensitive. Thus "a" matches "A", "á", "à", "ä", "Á", "À", "Â", and so on. (Note that in Norwegian, "ä" is treated as if it were "ae".)

  In Spanish, "ch" and "ll" are treated as if they were individual unique single letters. Thus, `CONTAINING "C"` will find "C", "c", "ç", and "Ç", but not "CH", "ch", "Ch" and "cH".

- MATCHING relational operator

  This operator is not sensitive to diacritical markings nor is it case sensitive. Thus, "a" matches "A", "á", "à", "ä", "Á", "À", "Â", and so on. (Note that in Norwegian, "ä" is treated as if it were "ae".)

  In Spanish, the combinations "ch" and "ll" are each treated as individual unique single letters. If you define your collating sequence as SPANISH, the percent sign (%) matches any single letter, including "ch" and "ll". "C%" and "C*" do not match "CH", "ch", "Ch", or "cH".

- STARTING WITH relational operator

  Because STARTING WITH is case sensitive, searches for uppercase multinational characters will not include lowercase multinational characters, and vice versa. For example, `STARTING WITH "Ç"` will retrieve a set of records that is different from those retrieved by `STARTING WITH "ç"`.

  In Spanish, "ch" and "ll" are treated as if they were individual unique single letters. For example, if a domain is defined with the collating sequence SPANISH, then `STARTING WITH "c"` will not retrieve the word "char", but it will retrieve the word "cat".

- Miscellaneous

  The character "ñ" is always treated as different from the character "n", in keeping with the practices of the Spanish language. In a similar manner, the character "ç" is treated the same as the character "c", in keeping with the practices of the French language.

  The character "ü" is treated the same as the character "u" for many languages, but is sorted between the characters "x" and "z" (with the "y"s) for Danish, Norwegian, and Finnish languages.

You can combine several conditional expressions by using a logical operator to form a compound conditional expression. A **logical operator** joins two or more conditional expressions together. The logical operators are:

- AND—Evaluates to true if all the conditions linked by the AND operator are satisfied.
- OR—Evaluates to true if at least one of the conditions linked by the OR operator is satisfied.
- NOT—Returns all other records in the record stream *except* those identified by the conditional expression following the NOT operator. Therefore, you retrieve the complement of the record stream identified by the RSE.

For information on using logical operators, see Section 3.5.2.

The WITH clause can be used in a variety of ways to limit the records returned in the record stream. The following sections illustrate some of the more commonly used ways.

3.5.1 Retrieving Records That Satisfy a Single Condition

To select only those records with a particular field value, specify a value for that field by including the WITH clause in the RSE. RDO retrieves only those records with the specified field value. For example:

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00246"
```

Here, the WITH clause tests whether or not there is a record with an employee identification number (employee ID) of 00246. As another example, you could request that RDO display the records of the EMPLOYEES relation that have a specific value for the CITY field, and RDO would retrieve only the records of employees living in the specified city.

3.5.2 Specifying Compound Conditions for Records

The preceding section described an RSE that includes a single conditional expression to test records. You can include more than one conditional test in the same RSE. The following sections show how to retrieve those records that satisfy a compound condition.

3.5.2.1 Retrieving Records That Satisfy Two or More Conditions: AND Operator When you want each record in the stream to satisfy two or more conditions, you can combine conditional expressions together with the AND logical operator. The record stream contains only those records that satisfy all the conditions within the compound conditional expression.

For example, to retrieve all part-time employees who live in Portsmouth, New Hampshire:

```
FOR E IN EMPLOYEES
 WITH E.CITY = "Portsmouth"
 AND E.STATE = "NH"
 AND E.STATUS_CODE = "2"
  PRINT
    E.EMPLOYEE_ID,
    E.CITY,
    E.LAST_NAME,
    E.STATUS_CODE
END_FOR
```

Table 3–2 shows how Rdb/VMS evaluates a compound conditional expression formed with the AND logical operator. A and B stand for simple conditional expressions that are components of the compound conditional expression, A AND B.

Table 3–2 AND Logical Operator

A	B	A AND B
True	False	False
True	True	True
False	False	False
False	True	False

(continued on next page)

Table 3–2 (Cont.) AND Logical Operator

A	B	A AND B
True	Missing	Missing
False	Missing	Missing
Missing	True	Missing
Missing	False	Missing
Missing	Missing	Missing

3.5.2.2 Retrieving Records That Satisfy One of Several Conditions: OR Operator

You may want to retrieve records that meet at least one of a series of conditions. To set up such a test, form a compound conditional expression with the OR logical operator. If any one of the component conditional expressions is true for a record, Rdb/VMS includes that record in the record stream.

The following compound conditional expression that uses the OR logical operator retrieves information about all employees who have received graduate degrees:

```
FOR D IN DEGREES WITH
 (D.DEGREE = "MA" ) OR
 (D.DEGREE = "PhD")
  PRINT
    D.DEGREE,
    D.EMPLOYEE_ID,
    D.COLLEGE_CODE,
    D.DEGREE_FIELD
END_FOR
```

To find the records for employees with either an MA degree or a PhD, specify two conditions linked by the logical operator OR. This means that RDO includes a record in the stream if either or both of the two conditions are true.

Note *You should enclose each conditional expression in parentheses and nest them to any level necessary to make the compound expression clear. Rdb/VMS evaluates the innermost expressions first and the outermost expressions last.*

Table 3–3 illustrates how Rdb/VMS evaluates a compound conditional expression formed with the logical operator OR. A and B stand for simple conditional expressions in the compound conditional expression, A OR B.

Table 3–3 OR Logical Operator

A	B	A OR B
True	False	True
True	True	True
False	True	True
False	False	False
True	Missing	Missing
False	Missing	Missing
Missing	True	Missing
Missing	False	Missing
Missing	Missing	Missing

3.5.2.3 Retrieving Records That Do Not Satisfy a Condition: NOT Operator

The third logical operator, NOT, enables you to retrieve records that are *not* identified by the conditional expression. You can include the NOT operator in a simple or compound conditional expression. RDO restricts the record stream to those records that do not satisfy the conditional expression following the NOT logical operator.

Table 3–4 illustrates how Rdb/VMS evaluates a compound conditional expression formed with the logical operator NOT. In this table, the first column represents a conditional expression A; the second column is the complement of A, or a condition that identifies all other records not identified by A.

Table 3–4 NOT Logical Operator

A	NOT A
True	False
False	True
Missing	Missing

You *cannot* use the NOT operator immediately preceding the following relational operators:

- EQ (=)
- NE (<>)
- GT (>)
- GE (>=)

- LT (<)
- LE (<=)

For example, you could not enter WITH S.SALARY_AMOUNT NOT = 30000. Instead, you could express that inequality in any of the following ways:

```
WITH NOT (S.SALARY_AMOUNT = 30000)
WITH S.SALARY_AMOUNT NE 30000
WITH S.SALARY_AMOUNT <> 30000
```

3.5.2.4 Using the ANY and NOT ANY Operators The ANY relational operator tests whether another record stream has any records (that is, is not empty). The following example shows a situation in which the ANY relational operator is useful:

```
FOR E IN EMPLOYEES WITH (ANY D IN DEGREES WITH
      D.EMPLOYEE_ID = E.EMPLOYEE_ID)
 PRINT
   E.EMPLOYEE_ID,
   E.LAST_NAME,
   E.FIRST_NAME
END_FOR
```

The ANY relational operator lets you refer to the records of a second relation, in this case, the DEGREES relation. Records from the first relation (EMPLOYEES) appear in the stream only when there is at least one record in the second relation (DEGREES) that meets the condition you specify (matching employee IDs). In evaluating this query, Rdb/VMS examines each record of the EMPLOYEES relation and compares the value of the EMPLOYEE_ID field with the same field in every record of the DEGREES relation.

Combining the NOT logical operator with the ANY relational operator allows you to specify that records from one relation appear in the record stream only when *no record* in another relation meets the condition you specify. The following example uses NOT ANY to retrieve records of employees who have no college degree:

```
FOR E IN EMPLOYEES
    WITH (NOT ANY D IN DEGREES
      WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID)
  PRINT
    E.EMPLOYEE_ID,
    E.LAST_NAME,
    E.FIRST_NAME
END_FOR
```

To access data about employees without college degrees, you need to access the EMPLOYEES and DEGREES relations. Each record from the EMPLOYEES relation is checked against the DEGREES relation to see if that employee's ID appears (that is, to see if the employee has a degree from some college). If an employee's ID does not appear in the DEGREES relation, you want to include the corresponding record from the EMPLOYEES relation in the stream.

In this example, the first WITH clause of the RSE in the FOR statement contains a second WITH clause to test and restrict the stream. The first WITH clause of the RSE is:

```
WITH NOT ANY D IN DEGREES
```

The second clause is:

```
WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
```

3.5.3 Retrieving Records That Match a Substring or Pattern

The MATCHING operator allows you to retrieve records based on a partial match of a field value. The MATCHING operator is not case sensitive. You can use the following wildcard characters in specifying the substring or pattern to be matched:

*	Matches any string of zero or more characters
%	Matches any single character

The MATCHING operator allows for more flexible searching than simple equality testing. For example, the following request finds all products where the COLOR field starts with “dark” and contains “red”:

```
FOR P IN PRODUCT WITH P.COLOR MATCHING "dark*red*"
   PRINT P.NAME,P.COLOR
END_FOR
```

The preceding might find product records with the following COLOR values:

Dark red
dark scarlet red
dark reddish brown

The following excerpts from an RDO terminal session log file show the results of combinations of substrings and matching characters:

```
!
! Find the first five names in which the letters "on" come
! last in the name. Note that the last "*" following a single
! space character is necessary because the field ends in an
! unknown number of spaces. (The REDUCED TO clause [explained
! in a later section] eliminates multiple appearances of a
! name if more than one employee has that last name.)
```

```
FOR FIRST 5 E IN EMPLOYEES WITH
    E.LAST_NAME MATCHING "*on *"
    REDUCED TO E.LAST_NAME
  PRINT
    E.LAST_NAME
END_FOR
 LAST_NAME
 Aaron
 Burton
 Clinton
 Dixon
 Ferguson

!
! Find the names in which the letters "on" come
! after the first character in the name.

FOR FIRST 5 E IN EMPLOYEES WITH
    E.LAST_NAME MATCHING "%on*"
    REDUCED TO E.LAST_NAME
  PRINT
    E.LAST_NAME
END_FOR
 LAST_NAME
 Connolly
 Jones
 Lonergan

!
! The MATCHING operator also works with numeric data types.
! Find the salaries that begin with the number 3. This
! is another way to find all the salaries in the
! range BETWEEN 30000 AND 39999 (if the minimum salary
! is 400 or higher, and the maximum is less than 300000).

FOR S IN SALARY_HISTORY WITH
 S.SALARY_AMOUNT MATCHING "3*"
  PRINT S.SALARY_AMOUNT
END_FOR
 SALARY_AMOUNT
 32254.00
 30598.00
 30880.00
 32589.00
 33944.00

!
! Find the salaries where the number 87 follows the
! first digit.
```

```
FOR S IN SALARY_HISTORY WITH
 S.SALARY_AMOUNT MATCHING "%87*"
  PRINT S.SALARY_AMOUNT
END_FOR
 SALARY_AMOUNT
 48797.00
 18705.00
 18778.00
 18778.00
 18746.00
```

3.5.4 Using Limited Matching: STARTING WITH and CONTAINING Operators

You can search for records in which a field value contains a specific sequence of characters. Two relational operators perform this type of search:

- STARTING WITH (case sensitive)
- CONTAINING (not case sensitive)

For example, assume you do not remember how to spell an employee's name, but you do know that the name begins with "Tol". To find and display the record for that employee, you could use the following query:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME STARTING WITH "Tol"
  PRINT
    E.FIRST_NAME,
    E.LAST_NAME,
    E.EMPLOYEE_ID
END_FOR
```

Use the STARTING WITH relational operator to search for records in the EMPLOYEES relation with a last name beginning with "Tol". The STARTING WITH relational operator, like EQUAL and NE, *is* case sensitive. If you ask for employees whose last names start with "TOL", RDO does not retrieve the record because the database stores the field value as "Tol", not "TOL".

You can use the CONTAINING relational operator for searches that are not case sensitive. If you substitute CONTAINING for STARTING WITH in the preceding example RDO will retrieve the Toliver record.

```
FOR E IN EMPLOYEES WITH E.LAST_NAME CONTAINING "TOL"
  PRINT
  E.FIRST_NAME,
  E.LAST_NAME,
  E.EMPLOYEE_ID
END_FOR
```

You can use the CONTAINING operator for searches on any part of a field value, not just the beginning. Thus, the CONTAINING operator is similar to the MATCHING operator; however, wildcard characters are not permitted with the CONTAINING operator (the * and % characters are treated as actual characters in the substring to be matched).

Note *Rdb/VMS does not use the index tables for indexed fields to evaluate conditional expressions that use the CONTAINING operator. If you are searching on the initial substring of a value for a field with an index defined, you can get better performance by using STARTING WITH instead of CONTAINING.*

3.6 Eliminating Duplicate Values: REDUCED TO Clause

Many records in a database contain fields that hold duplicate values. For example, many records in the EMPLOYEES relation may have “MA” stored in the STATE field. Some queries look for unique values for one or more fields in a record. For example, assume you want a list of the states in which employees live. If a state (such as Massachusetts, code MA) occurs more than once, you want it included only once.

The RSE in the following example finds the states in which all current employees live. Because many employees live in the same state, this query uses the REDUCED TO clause to restrict the final output to a unique value for the STATE field.

```
FOR E IN EMPLOYEES REDUCED TO E.STATE
  PRINT E.STATE
END_FOR
```

The preceding example forms a record stream from the EMPLOYEES relation, using an RSE with a REDUCED TO clause. The field named in the REDUCED TO clause (E.STATE) is called the *reduce key*. This clause eliminates any duplicate values for the field or combination of fields specified as reduce keys.

You can specify more than one reduce key. Assume you want to collect information about the range of colleges from which employees received degrees and the specific degree fields from each college. You need to display data about each college attended and the degrees granted by that college. If several employees attended the same college, display the college once; if several employees received the same type of degree from a college, display the degree data once. In other words, you want to restrict the stream to unique *combinations of values* for the college code and the degree. The following example requires two reduce keys, COLLEGE_CODE and DEGREE:

```
FOR D IN DEGREES REDUCED TO D.COLLEGE_CODE, D.DEGREE
 PRINT
   D.COLLEGE_CODE,
   D.DEGREE
END_FOR
```

In the preceding example, if two or more employees received PhD degrees from Stanford University, "COLLEGE_CODE = Stanford, DEGREE = PhD" would appear only once.

Be cautious in using the REDUCED TO clause if you are interested in fields other than the reduce key or keys. A query that specifies a REDUCED TO clause restricts the record stream by excluding duplicate records. In general, limit your display to those fields specified in the REDUCED TO clause. Displaying values of fields not specified in the REDUCED TO clause may yield unpredictable results. For example, if you specified CITY as a reduce key and yet also wanted to display employee identification numbers as well, the results can be misleading, as in the following example:

```
! This example shows a probable error in logic. It is not to
! be taken as a model of good coding.
FOR E IN EMPLOYEES REDUCED TO E.CITY
  PRINT
  E.CITY,
  E.EMPLOYEE_ID
END_FOR
```

In the preceding example, Rdb/VMS lists the unique occurrences of the CITY field, such as Keene, but it does not know which EMPLOYEE_ID field values you want (for example, which of the following residents of Keene: employee IDs 00186, 00219, 00230, or 00234). The EMPLOYEE_ID field value that RDO displays depends on how Rdb/VMS searches the records in the relation and selects a value for use. That search sequence can be different each time you execute the query.

The following table shows values for just three fields (CITY, STATE, and EMPLOYEE_ID) of a record as they actually occur in six hypothetical records of the EMPLOYEES relation. (The actual EMPLOYEES relation in the sample database contains 100 records, and none with these EMPLOYEE_ID values; however, for this illustration assume that the only records in the EMPLOYEES relation are those listed in the table.) The lists following the table show the effect of various REDUCED TO clauses.

Field values in the database:

CITY	STATE	EMPLOYEE_ID
Boston	MA	00123
Portsmouth	NH	00124
New Bedford	MA	00125
Manchester	NH	00126
Boston	MA	00127
Portsmouth	RI	00128

Reduced to CITY:

Boston
Portsmouth
New Bedford
Manchester

Reduced to STATE:

MA
NH
RI

Reduced to CITY, STATE:

Boston, MA
Portsmouth, NH
New Bedford, MA
Manchester, NH
Portsmouth, RI

3.7 Testing for a Single Record Occurrence: UNIQUE Operator

You can test a relation to determine the uniqueness of a record occurrence. A record is unique if there is exactly one record that satisfies the RSE. You can find a unique record by specifying a field whose value makes that record unique. For example, assume that you wish to display any city in which only one employee lives, along with the name of the employee living there:

```
FOR E IN EMPLOYEES
  WITH UNIQUE EMP IN EMPLOYEES
  WITH E.CITY = EMP.CITY
   PRINT E.CITY,
     E.LAST_NAME,
     E.FIRST_NAME
END_FOR
```

Use NOT UNIQUE to find instances where more than one record satisfies the RSE. For example, you can modify the preceding example to display only those cities in which two or more employees live, along with the employees' names:

```
FOR E IN EMPLOYEES
  WITH NOT UNIQUE EMP IN EMPLOYEES
  WITH E.CITY = EMP.CITY
   PRINT E.CITY,
     E.LAST_NAME,
     E.FIRST_NAME
END_FOR
```

The UNIQUE operator differs from the REDUCED TO clause in one important way. When you use the REDUCED TO clause, you should display only the values of the fields named in that clause; displaying other field values will produce unanticipated results. The UNIQUE operator locates an entire record whose field value makes the record unique and allows you to display any or all fields from the qualifying record.

Table 3–5 summarizes the effects of the ANY, NOT ANY, UNIQUE, and NOT UNIQUE operators.

Table 3–5 Testing for the Existence of Records with ANY and UNIQUE Operators

Operator Preceding RSE	True if:
ANY	At least one record found
NOT ANY	No records found
UNIQUE	Only one record found
NOT UNIQUE	More than one record found, or no records found

3.8 Retrieving Segmented Strings

The **segmented string** is a special Rdb/VMS data type designed to handle large pieces of data with a segmented internal structure. The maximum size of an individual string segment is 64K bytes. Except for the length of the string's segments, Rdb/VMS does not know anything about the type of data contained in a segmented string. For example, in a segmented string you might store large amounts of text, long strings of binary input from a data collecting device, or graphic data. A program can then retrieve the data from the database and handle it in the appropriate way.

Because Rdb/VMS does not know what kind of data is contained in a segmented string, you cannot perform many of the standard data manipulation functions on it. You cannot use relational operators, such as EQUAL and CONTAINING, to compare segmented strings. Rdb/VMS does not perform any data type conversion on data that is transferred into or out of a segmented string.

Rdb/VMS cannot modify a segmented string; it can only read or write it. Therefore, to modify a segmented string, you must read it, make any changes, and then write out the new segmented string.

Rdb/VMS defines a symbol to refer to the segments of a segmented string. This symbol is equivalent to a field name; it names the "fields" or segments of the string. Furthermore, because segments can vary in length, Rdb/VMS also defines a symbol for the length of a segment. You must use these symbols in the value expressions that you use to retrieve the length and value of a segment:

- RDB$VALUE

  The value stored in a segment of a segmented string
- RDB$LENGTH

  The length in bytes of a segment

Because a single segmented string field value is made up of multiple segments, you must manipulate the segments one at a time. Therefore, segmented string operations require an internal looping mechanism, much like the record stream set up by a FOR or START_STREAM statement. The following example retrieves and prints two segmented strings:

```
FOR R IN RESUMES WITH R.EMPLOYEE_ID = '00164'
  FOR S IN R.RESUME
     PRINT S.RDB$LENGTH, S.RDB$VALUE
  END_FOR
END_FOR
```

4 Retrieving Records and Joining Relations

You can use relational operators and conditional expressions to retrieve records from a single relation or from several relations joined together. This chapter illustrates how to join relations to retrieve information contained in more than one relation.

4.1 Joining Relations Using the CROSS Clause

Sometimes you must look at two or more relations to find the information that satisfies a query.

The need to join relations to satisfy a query is a by-product of the process of database normalization. When you design a relational database, you try to *normalize* it by dividing groups of data elements into separate relations. Common fields in each relation link one relation with another. Normalization helps you avoid storing redundant data. (See the *VAX Rdb/VMS Guide to Database Design and Definition* for details and examples of normalization.)

For example, you need not store information about each job an employee has held in the company with employee information. You can store employee information in an EMPLOYEES relation and job history information in a JOB_HISTORY relation. The relations share a common field: EMPLOYEE_ID. When you need to retrieve information about a worker and his or her job history, you *join* the two relations on the EMPLOYEE_ID field.

You can join one relation with another, or you can join one relation with itself. The following sections describe the variations.

4.1.1 Joining Records from Two Relations

The simplest type of join combines records from two relations that have a matching value for a common field. Consider a query to find the job history and related job information for employees.

You first look at the JOB_HISTORY relation and find that it has six fields:

- EMPLOYEE_ID (the employee's identification number)
- JOB_CODE (the employee's job code)
- JOB_START (the employee's starting date)
- JOB_END (the date the employee ended the job)
- DEPARTMENT_CODE (the employee's department)
- SUPERVISOR_ID (the identification number of the employee's supervisor)

The JOB_HISTORY relation, however, does not tell you all you need to know about an employee's job, because you also need to know the person's job title, wage class, and the minimum and maximum salaries for that person's job. This additional information is found in a separate relation named JOBS.

You can combine information from both relations with one query, displaying all the data as though it were one record. To do this, join a record from the JOB_HISTORY relation with a corresponding record from the JOBS relation. The CROSS clause of the RSE enables you to *cross* or join records using a field common to both relations, JOB_CODE. The WITH clause specifies that only those records with a JOB_CODE value in one relation that matches a JOB_CODE value in the other relation are joined. Those records in one relation with JOB_CODE values that do not match JOB_CODE values in the other relation are excluded from the join. Because this clause joins related records from two relations, it is a **relational join**.

The following example joins records from the JOB_HISTORY and JOBS relations, printing the requested information for each record in the JOB_HISTORY relation:

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS
    WITH JH.JOB_CODE = J.JOB_CODE
 PRINT
     JH.EMPLOYEE_ID,
     JH.DEPARTMENT_CODE,
     JH.JOB_CODE,
     J.WAGE_CLASS,
     J.JOB_TITLE,
     J.MINIMUM_SALARY,
     J.MAXIMUM_SALARY
END_FOR
```

When you join two or more relations in this way, you form an expanded output record that contains data from several associated relations.

Note that although the common field might have the same name in both relations, it does not have to; that is, the field name in each relation will contain the same type of information, but the field can have a different name in each relation. For example, the sample database contains the fields EMPLOYEE_ID, MANAGER_ID, and SUPERVISOR_ID. All of these fields contain employee identification numbers, but some identification numbers serve different purposes. However, any of these fields can serve as a join term linking two relations together. The **join terms** contain logically identical data that you can use to link relations in a join.

If the fields do have the same name, you can use an OVER clause with the CROSS clause to specify the join term. In the following example, CROSS J IN JOBS OVER JOB_CODE is equivalent to CROSS J IN JOBS WITH JH.JOB_CODE = J.JOB_CODE in the previous example:

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS OVER JOB_CODE
 PRINT
     JH.EMPLOYEE_ID,
     JH.DEPARTMENT_CODE,
     JH.JOB_CODE,
     J.WAGE_CLASS,
     J.JOB_TITLE,
     J.MINIMUM_SALARY,
     J.MAXIMUM_SALARY
END_FOR
```

Using a WITH or OVER clause to qualify or limit a join lets you link related records from two relations. Although Rdb/VMS can process queries without the WITH or OVER clauses, the results are not very meaningful. You should include either clause in every join. If you do not specify a WITH or OVER clause, RDO joins each record of one relation with every record of the other relation, giving you a cross product. Such a join can be disastrous to your system's performance.

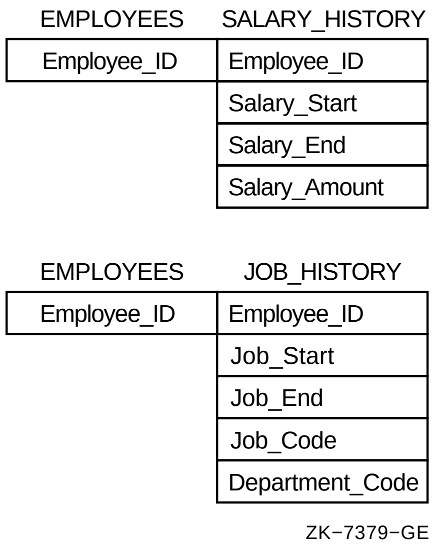
For example, if you join the EMPLOYEES relation and the JOB_HISTORY relation without qualifying the relationship, Rdb/VMS joins every record in the EMPLOYEES relation with *every* record in the JOB_HISTORY relation. If there were 101 records in the EMPLOYEES relation and 277 records in the JOB_HISTORY relation, the cross product would contain 27,977 records.

4.1.1.1 One-to-One and One-to-Many Joins

If one relation contains unique key values for each record, you can easily join this relation with another relation that contains either similar unique key values for each record or multiple records with the same key value. Such a relationship is either one-to-one or one-to-many.

In the EMPLOYEES relation, EMPLOYEE_ID is a key field that contains a unique value for each record in the relation. The JOB_HISTORY and SALARY_HISTORY relations contain many records that belong to an individual employee, because one employee can have many JOB_HISTORY records and many SALARY_HISTORY records. You can join the EMPLOYEES relation with the JOB_HISTORY relation to find all records that belong to a single employee. You can also join the EMPLOYEES relation with the SALARY_HISTORY relation to retrieve all salary history records for that employee. The results of such joins are illustrated in Figure 4–1; you assemble employee information with every JOB_HISTORY record or with every SALARY_HISTORY record.

Figure 4–1 One-to-Many Joins of Relations



4.1.1.2 Many-to-Many Joins The joins in Figure 4–1 (of EMPLOYEES with SALARY_HISTORY and of EMPLOYEES with JOB_HISTORY) are straightforward. However, joining the JOB_HISTORY relation with the SALARY_HISTORY relation presents a special situation. Each of these relations contains multiple records for an employee; such a relationship is a many-to-many relationship.

When you attempt a join with a many-to-many relationship, the cross product is likely to be meaningless. If you join the JOB_HISTORY relation with the SALARY_HISTORY relation using the join term EMPLOYEE_ID, Rdb/VMS joins every record from the JOB_HISTORY relation for an employee with *every* record from the SALARY_HISTORY relation. Thus, every JOB_HISTORY record is associated with every SALARY_HISTORY record for a particular employee. But the relationship you need should be qualified further by checking date values in these records to link a JOB_HISTORY record with the SALARY_HISTORY record for *that particular job*.

To ensure that your join produces the correct results, use the AND operator within the WITH clause to qualify the join terms precisely. The following query contains a CROSS clause that joins the JOB_HISTORY relation with the SALARY_HISTORY relation and qualifies the join by including several AND operator statements in the WITH clause:

```
FOR JH IN JOB_HISTORY
  CROSS SH IN SALARY_HISTORY
   WITH JH.EMPLOYEE_ID = SH.EMPLOYEE_ID
    AND JH.JOB_END MISSING
    AND SH.SALARY_END MISSING
    AND JH.EMPLOYEE_ID = "00164"
        PRINT
                JH.EMPLOYEE_ID,
                JH.JOB_CODE,
                JH.JOB_START,
                SH.SALARY_START,
                SH.SALARY_AMOUNT,
                SH.SALARY_END
END_FOR
```

The following information is displayed:

```
00164   DMGR   21-SEP-1981  21-SEP-1981   50000.00  17-NOV-1858
```

The results of this query restrict the records retrieved from the JOB_HISTORY and SALARY_HISTORY relations as follows:

- The employee ID is 00164.
- Only records from both relations belonging to employee ID 00164 are retrieved.

- Only current records from the JOB_HISTORY and SALARY_HISTORY relations are needed; that is, the fields JOB_END and SALARY_END are missing because you are looking for employees who have not terminated their employment.

In some instances, it may be useful to add a condition that contains obvious information but is helpful to the Rdb/VMS query optimizer. If you added the condition DEPARTMENT_CODE NOT MISSING to the preceding RSE, and DEPARTMENT_CODE were an indexed field, the query optimizer would process the query more efficiently. For more information about this process, see the *VAX Rdb/VMS Guide to Database Maintenance and Performance*.

4.1.2 Joining Records from More Than Two Relations

When you join two relations at a time, you need a CROSS clause for the pair of relations in the join. If you need to retrieve data from three relations, first join records from two relations on one field (join term); then join one relation of the first pair with the third relation, using either the same join term or a different one.

Previously, you saw a query that joined two relations, JOB_HISTORY and JOBS, to retrieve all information about a specific job. However, assume you also need to find the full name of an employee. To associate an employee with the full name information (first name and last name), you expand the query with another CROSS clause to access the EMPLOYEES relation and to supply an employee ID number, as in the following example:

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS OVER JOB_CODE
    CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
      WITH E.EMPLOYEE_ID = "00164"
  PRINT
    JH.EMPLOYEE_ID,
    E.FIRST_NAME,
    E.LAST_NAME,
    J.JOB_TITLE,
    JH.DEPARTMENT_CODE,
    J.WAGE_CLASS
END_FOR
```

The preceding query adds a second CROSS clause to get the necessary information from the EMPLOYEES relation. With this query, you access the following three relations:

- JOBS (data about each type of job)
- JOB_HISTORY (data about each job held by each employee)
- EMPLOYEES (personal data about each employee)

You join the JOB_HISTORY and JOBS relations on the JOB_CODE field to get complete job information. Then, to include the employee data associated with each set of job history records, you join the JOB_HISTORY relation with the EMPLOYEES relation. To ensure that the job data corresponds to the correct employee, you perform this second join on the EMPLOYEE_ID field. To join related records from three relations, Rdb/VMS:

- Forms a stream that combines records from the JOBS and JOB_HISTORY relations with matching values for the JOB_CODE field.
- Adds records from the EMPLOYEES relation that restrict values to that of EMPLOYEE_ID = "00164" for the records in the record stream of the first join. Now, the complete join operation includes only those records belonging to employee 00164.

Each resulting output record has fields from the three relations: JOBS, JOB_HISTORY, and EMPLOYEES. Each CROSS clause uses a join term common to two relations:

- JOB_CODE is the join term for the JOB_HISTORY and JOBS relations.
- EMPLOYEE_ID is the join term for the EMPLOYEES and JOB_HISTORY relations.

To improve the efficiency of complex joins, you can define an index for each frequently used join term. See the *VAX Rdb/VMS Guide to Database Design and Definition* for information on defining indexes. When your query requires joins of two or more relations, you can include the names of the relations in the RESERVING clause of your START_TRANSACTION statements. (If you do not use the RESERVING clause, all of the relations are automatically reserved when they are referenced.) For example, the query illustrated earlier in this section refers to three relations: EMPLOYEES, JOBS, and JOB_HISTORY. Your START_TRANSACTION statement might look like the following:

```
RDO> START_TRANSACTION READ_ONLY RESERVING
cont>             EMPLOYEES FOR SHARED READ,
cont>             JOBS FOR SHARED READ,
cont>             JOB_HISTORY FOR SHARED READ
```

4.1.3 Joining One Relation on Itself

Another type of query, called a **reflexive join**, allows you to join records from one relation with other records in the same relation. You treat the relation as if it were actually two relations, supplying two different context variables in the join. Perform a reflexive join when you wish to match values from fields of the same relation.

For example, you might want to list all job classifications of staff employees whose maximum salaries are greater than the minimum salaries of company executives. The JOBS relation contains information about jobs in all wage classes. Staff members are identified as wage class 2 and executives as wage class 4.

You could access the JOBS relation two separate times, first retrieving all maximum salaries for wage class 2, and then retrieving all minimum salaries for wage class 4; you would then compare the records to find where the salaries overlapped. However, not all of these steps are necessary, because Rdb/VMS lets you access the relation twice in the same query by means of a reflexive join, as shown in the following example:

```
FOR EXEC IN JOBS
    CROSS STAFF IN JOBS
       WITH EXEC.WAGE_CLASS = "4"
       AND STAFF.WAGE_CLASS = "2"
       AND STAFF.MAXIMUM_SALARY > EXEC.MINIMUM_SALARY
   PRINT
       STAFF.JOB_CODE,
       STAFF.MAXIMUM_SALARY,
       EXEC.JOB_CODE,
       EXEC.MINIMUM_SALARY
END_FOR
```

Note *When you use descriptive context variables like STAFF and EXEC, you are more likely to refer to the field names correctly. You know at a glance the stream to which you are referring.*

The preceding RDO query joins the JOBS relation on itself. The query specifies two different context variables, STAFF and EXEC, for the same relation, JOBS. These statements instruct RDO to form a stream that includes records containing data on pairs of employees, STAFF and EXEC.

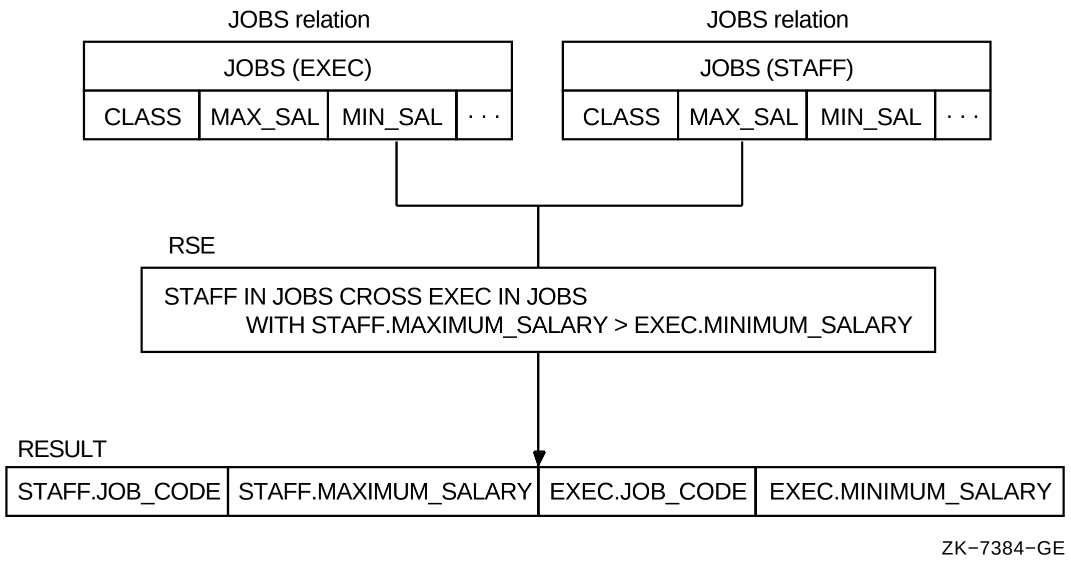
To process this query, Rdb/VMS:

- Takes the first record in STAFF (wage class = 2) and compares the maximum salary amount with the minimum salary amount of the first record in EXEC (wage class = 4).
- Compares the first record in STAFF with the next record of EXEC until all EXEC records have been compared. Rdb/VMS makes one pass through EXEC for each record in STAFF.
- Takes the second record in STAFF and compares it to the first record in EXEC.

- Compares the second record in STAFF with the second record of EXEC, and so on.
- Includes in the resulting record stream only those records that meet the specified conditions.

As Figure 4–2 illustrates, the JOBS relation appears as two relations: STAFF and EXEC.

Figure 4–2 Joining a Relation on Itself (Reflexive Join)



4.2 Using Nested FOR Loops

When you use the CROSS clause to join two relations in a one-to-many relationship, Rdb/VMS links the record in the first relation to each record in the second relation. All values from the first relation are present in the resulting cross, but only the values common to both the first and second relation are included from the second relation.

The following two examples request the same information; however, the second uses a nested FOR loop to make the display more readable:

```
FOR E IN EMPLOYEES
  CROSS JH IN JOB_HISTORY
    WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID
    AND E.EMPLOYEE_ID = "00201"
      PRINT
       E.EMPLOYEE_ID,
       JH.JOB_CODE,
       JH.JOB_START,
       JH.JOB_END
END_FOR
```

```
 00201   APGM   15-APR-1979 00:00:00.00   27-MAY-1980 00:00:00.00
 00201   APGM   28-MAY-1980 00:00:00.00   17-NOV-1858 00:00:00.00
 00201   APGM    1-JUL-1975 00:00:00.00    3-JUN-1977 00:00:00.00
 00201   APGM    4-JUN-1977 00:00:00.00   14-APR-1979 00:00:00.00
```

Note that 17-NOV-1858 is the VMS base date, used here to indicate a missing value for fields with a DATE data type.

The next example uses a nested FOR loop to display the employee ID only once, rather than with each job history record for that employee. This display is a simple example of a *control break report*. Note that while the format may be more attractive, using a nested FOR loop instead of a join may limit the query optimizer's ability to select the most efficient retrieval strategy.

```
FOR E IN EMPLOYEES
  WITH E.EMPLOYEE_ID = "00201"
    PRINT E.EMPLOYEE_ID

      FOR JH IN JOB_HISTORY
        WITH JH.EMPLOYEE_ID = E.EMPLOYEE_ID

         PRINT
          JH.JOB_CODE,
          JH.JOB_START,
          JH.JOB_END
    END_FOR

END_FOR
```

```
 00201
       APGM    1-JUL-1975 00:00:00.00    3-JUN-1977 00:00:00.00
       APGM   28-MAY-1980 00:00:00.00   17-NOV-1858 00:00:00.00
       APGM   15-APR-1979 00:00:00.00   27-MAY-1980 00:00:00.00
       APGM    4-JUN-1977 00:00:00.00   14-APR-1979 00:00:00.00
```

Nesting FOR loops means entering one FOR statement (the outer loop) followed by a second FOR statement (the inner loop). The inner loop is part of the main FOR statement that controls it. Each loop has an RSE to bring the two record streams together in the same statement.

The process in the preceding (second) example works as follows:

- RDO retrieves the first record in the stream formed by the outer loop and displays any expressions listed in the PRINT statement.
- RDO then processes the inner loop for each record specified by the inner loop's RSE.
- Control returns to the outer loop, and the cycle continues until there are no more records in the outer loop's stream.

With a nested FOR format, you display a value for the EMPLOYEE_ID field only when that field's value changes. Nested FOR loops are convenient for this type of display because you can show a one-to-many relationship: one EMPLOYEES record to many JOB_HISTORY records. You can also use nested FOR loops to establish relationships for outer joins.

5 Defining and Using Views

A **view** can be thought of as a virtual relation, a combination of fields from one or more relations in the database, that sometimes includes an RSE. A view does not contain actual (physical) records; rather, a view is a database element that provides flexibility in accessing data in records from one or more relations. A view can contain clauses limiting the record stream based on certain field values (such as WITH E.STATUS_CODE = 1). You can treat a view as you would a relation in creating RSEs.

If a view definition refers to a single relation, you can use it to perform the full range of data manipulation operations (read-only and read/write). However, if a view definition refers to multiple relations (or refers to another view that refers to multiple relations), you are limited to read-only access to the fields in the view definition.

Views provide the following benefits:

- View definitions allow you to combine fields used in queries that are executed frequently. This saves keystrokes and reduces the chance of error when users enter these queries.
- View definitions can prevent unauthorized users from accessing sensitive data, while still allowing users to access the data they need. This is done by creating different views for use by different classes of users; for example, create one view for general user access that omits fields with sensitive information from the view definition, and create another view for restricted user access that includes sensitive information.
- Queries that use complex selection criteria can be formalized in a view definition to make access easy.

5.1 Creating the View Definition

After some experience using your database, you may discover that users often enter the same query to display certain fields. Or you may decide that certain users, for security reasons or for reasons of job efficiency, need to see a subset of fields or a collection of fields from separate relations. You can use the DEFINE VIEW statement to create view definitions that meet the needs of all these users.

For example, you may discover that you often use some fields in the EMPLOYEES relation more than you use others. You can define a view to create a more restricted version of the EMPLOYEES relation using fields from that relation. The following definition creates a view that consists of the last name, first name, and employee ID. (Because you are creating a new definition in the database, you need to include write access to the database in your START_TRANSACTION statement or accept the default access of read/write.)

```
START_TRANSACTION READ_WRITE
```

```
DEFINE VIEW EMP_ID OF E IN EMPLOYEES.
  E.LAST_NAME.
  E.FIRST_NAME.
  E.EMPLOYEE_ID.
END VIEW.
```

```
COMMIT
```

Now you can refer to the view just as you refer to a relation, using the same field names as in the EMPLOYEES relation. For example:

```
FOR E IN EMP_ID
 PRINT E.*
END_FOR
```

You can also include an RSE when you refer to a view to restrict the records you display. For example:

```
FOR E IN EMP_ID WITH E.EMPLOYEE_ID = "00164"
 PRINT E.*
END_FOR
```

The preceding example displays the fields in the EMP_ID view for the record in which EMPLOYEE_ID = “00164”. If you forget which fields are included in the view definition, first use the SHOW RELATIONS statement, because views are considered to be relations. RDO displays all relations and indicates which definitions are views. For example:

```
RDO> SHOW RELATIONS

User Relations in Database with filename  personnel
     CANDIDATES
     COLLEGES
     CURRENT_INFO                     A view.
     CURRENT_JOB                      A view.
     CURRENT_SALARY                   A view.
     DEGREES
     DEPARTMENTS
     EMPLOYEES
     JOBS
     JOB_HISTORY
     RESUMES
     SALARY_HISTORY
     WORK_STATUS
```

You can then use the SHOW FIELDS statement to display the names of the fields in the CURRENT_INFO definition:

```
RDO> SHOW FIELDS FOR CURRENT_INFO

 Fields for relation  CURRENT_INFO
     LAST_NAME                        text size is  14
     FIRST_NAME                       text size is  10
     ID                               text size is  5
       based on global field  ID_NUMBER
     DEPARTMENT                       text size is  30
       based on global field  DEPARTMENT_NAME
     JOB                              text size is  20
       based on global field  JOB_TITLE
     JSTART                           Date
       based on global field  STANDARD_DATE
     SSTART                           Date
       based on global field  STANDARD_DATE
     SALARY                           signed longword scale  -2
```

You can limit the view to records that meet certain criteria based on field values. The following example defines a view named ACTIVE_EMP, limiting it to records where the status code is “1” (that is, full-time employees) and specifying the fields to be included.

```
DEFINE VIEW ACTIVE_EMP OF E IN EMPLOYEES
  WITH E.STATUS-CODE = "1".
   E.EMPLOYEE_ID.
   E.LAST_NAME.
   E.FIRST_NAME.
   E.MIDDLE_INITIAL.
   E.ADDRESS_DATA_1.
   E.ADDRESS_DATA_2.
   E.CITY.
   E.STATE.
   E.POSTAL_CODE.
   E.SEX.
   E.BIRTHDAY.
   E.STATUS_CODE.
END VIEW.
```

5.2 Joining Relations in a View Definition

The design of the PERSONNEL database includes many relations. Some applications may require a view of the database that combines fields from many relations. Rdb/VMS lets you create new relationships from the basic relations in the database by defining views using a join statement.

If you must often form the same RSE to retrieve records from several relations, you might consider creating a view definition. A view brings together fields from several relations based on an RSE specified in the view definition. A user can refer to the view definition as if it were a single relation and request RDO to display field values. Thus, a user who may not understand the syntax for a complex join can still access data from such a join when it is contained in a view.

The following example defines the view named EMP_HISTORY, which contains fields from three relations (JOB_HISTORY, JOBS, and EMPLOYEES). The view definition consists of two parts:

- A record selection expression (RSE)
- A list of the fields

```
DEFINE VIEW EMP_HISTORY
   OF JH IN JOB_HISTORY
       CROSS J IN JOBS
       CROSS E IN EMPLOYEES
       WITH J.JOB_CODE = JH.JOB_CODE AND
       JH.EMPLOYEE_ID = E.EMPLOYEE_ID.
          JH.EMPLOYEE_ID.
          E.FIRST_NAME.
          E.LAST_NAME.
          J.JOB_TITLE.
          JH.DEPARTMENT_CODE.
          J.WAGE_CLASS.
END VIEW.
```

You can now use the view definition EMP_HISTORY with an RSE to retrieve employee history data for a particular employee, as in the following example. You can include the name of the view in a START_TRANSACTION statement. Rdb/VMS implicitly reserves the relations referred to in the view according to the specified reserving option or its default.

```
START_TRANSACTION READ_ONLY RESERVING EMP_HISTORY FOR SHARED READ
FOR E IN EMP_HISTORY WITH E.EMPLOYEE_ID = "00164"
    PRINT E.*
END_FOR
```

The next example shows a more complex view that joins four relations. Assume that you need to compile a report for each employee in the EMPLOYEES relation, and the report should include the following information:

- Each employee's ID
- Each employee's first and last name
- The title of the job he or she currently holds
- The code of the department where the employee works
- Current salary information
- The wage class of the job the employee holds

The SALARY_HISTORY relation contains current salary information as well as the salary start date and salary amount for a particular job. The JOB_ HISTORY relation holds data about each job an employee has held, including the department and job code. The JOBS relation contains information about each job in the company. The EMPLOYEES relation describes each employee in the company. Each of these relations supplies some data for the report. To get the necessary fields from each, you must join the four relations in a view definition, as shown in the next example.

```
DEFINE VIEW EMPLOYEE_REPORT_DATA
          OF SH IN SALARY_HISTORY
        CROSS J IN JOBS
        CROSS E IN EMPLOYEES
        CROSS JH IN JOB_HISTORY
                WITH SH.SALARY_END MISSING
                AND SH.EMPLOYEE_ID = E.EMPLOYEE_ID
                AND JH.EMPLOYEE_ID = E.EMPLOYEE_ID
                AND JH.JOB_END MISSING
                AND JH.JOB_CODE = J.JOB_CODE.
                SH.EMPLOYEE_ID.
                E.FIRST_NAME.
                E.LAST_NAME.
                J.JOB_TITLE.
                JH.DEPARTMENT_CODE.
                J.WAGE_CLASS.
                SH.SALARY_AMOUNT.
                SH.SALARY_START.
END VIEW.
```

The preceding view definition refers to the following relations:

- JOB_HISTORY (each job ever held by each employee)
- JOBS (each type of job an employee can hold)
- EMPLOYEES (personal information on each employee)
- SALARY_HISTORY (each salary level held by an employee for each job held by each employee)

To define the view in the preceding example, you need to:

1 Identify the four relations with enough CROSS clauses to specify those relations that contain the fields you want to compare or display.

2 Add the WITH clauses to link each pair of relations that share a common field or meet a specific condition. One relation can be associated with more than one other relation:

    - The JOB_HISTORY relation links with the SALARY_HISTORY relation in the SALARY_END field.
    - Records with the JOB_END field containing the missing value are current jobs. (For information on the *missing value* for a field, including using the MISSING keyword to check if a field value is missing, see Section 6.5.) The JOB_HISTORY relation also links with the JOBS relation on the JOB_CODE field.
    - The SALARY_HISTORY relation links with the EMPLOYEES relation on the EMPLOYEE_ID field, but only for those records in the SALARY_HISTORY relation whose SALARY_END date field contains the missing value for the job held currently.

You can use any field in one relation that is common to any other relations. Each WITH clause links a field in one relation with a field in another and further restricts the records included in your record stream.

3 Display only those fields you need. Qualify each field with its context variable.

6 Storing, Modifying, and Erasing Data

This chapter shows you how to use RDO to store, modify, and erase data in an Rdb/VMS database. After you become familiar with the statements that perform these operations, you can include them in your host language programs: use the EDIT statement, write a series of statements to a file, and include that file in a host language source program. (See the *VAX Rdb/VMS RDO and RMU Reference Manual* for information on the EDIT statement.)

6.1 Storing Data in an Rdb/VMS Database

After you define your database and its various elements, you can store data in each relation using one of the following methods:

- Use the RDO STORE statement.
- Use the SQL INSERT statement.
- Embed your store operations in a host language program to load data into your database interactively or from disk files.
- Use DATATRIEVE to load data from VMS RMS files or a VAX DBMS database.
- Use a RALLY application, DECdecision, or TEAMDATA to enter data.

This section discusses only Rdb/VMS RDO storage statements.

6.1.1 Storing Values in One Relation

Entering and maintaining data in a database is an ongoing task. For example, every time a new employee joins a particular company, the Personnel department adds a record to a relation such as the EMPLOYEES relation described throughout this guide. Adding a new record to a relation in the database does not present a record-level conflict to other users of the database because they cannot access a record that does not yet exist. The START_TRANSACTION statement that precedes the STORE statement should specify the SHARED WRITE reserving option:

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

Setting the share mode to SHARED ensures that you are able to store several records in a session while other users access the same relation. However, other users can specify transaction modes in their START_TRANSACTION statements that conflict with your intentions; and when such conflicts occur, RDO might not allow other users access to the relation until you terminate your transaction. Refer to Chapter 2 for details on access conflicts.

When an employee joins the company, you should have enough information to store values for each field in a record of the EMPLOYEES relation. In Rdb/VMS, you use the STORE statement to insert a record into a relation. This statement usually includes a series of assignments that specify the values for each field of the record. For example, if the EMPLOYEES relation had only four fields, the following example would store a new record:

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

STORE E IN EMPLOYEES USING
      E.EMPLOYEE_ID = "00502";
      E.FIRST_NAME = "Paul";
      E.LAST_NAME = "Chris";
      E.CITY = "Boston"
END_STORE

COMMIT
```

6.1.2 Storing Values in Multiple Relations

You can store values in more than one relation within the same FOR statement in RDO. However, you need a STORE statement for each relation. For example, after you have entered all the records for new employees in the EMPLOYEES relation, you may want to add corresponding employee records in the JOB_HISTORY and SALARY_HISTORY relations. No field in the EMPLOYEES relation distinguishes a new employee record from existing employee records. You can, however, use the EMPLOYEES relation to compare existing EMPLOYEE_ID values with those already stored in the other two relations.

The JOB_HISTORY and SALARY_HISTORY relations should have at least one corresponding record for every employee currently working in the company. Therefore, those records in the EMPLOYEES relation with no records in the JOB_HISTORY or SALARY_HISTORY relations must be newly hired employees. You can perform a join across these three relations to check for the existence of corresponding records. If there are no matching records in the history relations, you can store new records in the history relations using values from the EMPLOYEES relation.

Before you store new values in the database, you should specify the necessary relations in your START_TRANSACTION statement and indicate the correct share mode and lock type. Notice you need write access to the JOB_HISTORY and SALARY_HISTORY relations, but only read access to the EMPLOYEES relation.

```
START_TRANSACTION READ_WRITE RESERVING
        EMPLOYEES FOR SHARED READ,
        JOB_HISTORY FOR SHARED WRITE,
        SALARY_HISTORY FOR SHARED WRITE
```

In the following example, the use of the NOT ANY relational operator and the AND logical operator in the WITH clause allows you to check that no corresponding records exist in the history relations. That is, RDO executes the STORE statement only when there is no such correspondence. This example stores data in the JOB_HISTORY and SALARY_HISTORY relations for each new employee and also prints out that person's EMPLOYEE_ID value.

```
FOR E IN EMPLOYEES
  WITH
    (NOT ANY JH IN JOB_HISTORY
      WITH JH.EMPLOYEE_ID = E.EMPLOYEE_ID)
  AND
    (NOT ANY SH IN SALARY_HISTORY
      WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID)

        PRINT E.EMPLOYEE_ID

        STORE JH IN JOB_HISTORY USING
          JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
          JH.JOB_START = "21-AUG-1989"
        END-STORE

        STORE SH IN SALARY_HISTORY USING
          SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
          SH.SALARY_START = "21-AUG-1989"
        END-STORE

END_FOR
```

You can include multiple STORE statements like these in a single transaction, and it is in fact good practice to keep related updates together in the same transaction.

6.2 Modifying Data

When you update data in a relation, you first must identify the record stream that contains the record or records you want to change. You can assign new values, use existing values in fields from other relations, or specify value expressions to calculate the new value for each field in the selected records of the record stream. After you create a record stream, Rdb/VMS executes the MODIFY statement to change the values of the specified field or fields.

You can modify data in more than one relation in a single transaction. If the changes to records in these relations depend on the values of fields in another relation (such as the EMPLOYEES relation), you must specify a record stream that contains the necessary records from that other relation.

When you are modifying relations, you can choose to let other users access those same relations. But once you select a record to change, Rdb/VMS locks that record. Other users cannot access that record until you release it. The lock ensures that only current data is available to all users. The changes you make are available to other users only after you make your changes permanent with the COMMIT statement. If you decide that the changes you make should not apply to the database at this time, you can undo the updates with the ROLLBACK statement.

6.2.1 Changing Data in a Single Relation

The simplest form of data modification involves changing data in a single relation.

The following example uses an RSE that identifies all people currently employed in the Engineering department. It finds the records of all current employees in the JOB_HISTORY relation using the JOB_END MISSING expression. When the records are selected, the RSE uses the MODIFY statement to change all the supervisor identification numbers in Engineering to 00348.

```
START_TRANSACTION READ_WRITE RESERVING JOB_HISTORY
        FOR SHARED WRITE

FOR JH IN JOB_HISTORY
   WITH JH.DEPARTMENT_CODE = "ENG "
   AND JH.JOB_END MISSING

      MODIFY JH USING
        JH.SUPERVISOR_ID = "00348"
      END_MODIFY

END_FOR
COMMIT
```

You can check the effect of the changes you just made. The following example displays information about the employees currently in the Engineering department, including the new supervisor ID:

```
RDO> FOR JH IN JOB_HISTORY
cont>    WITH JH.DEPARTMENT_CODE = "ENG"
cont>    AND JH.JOB_END MISSING
cont>       PRINT
cont>         JH.EMPLOYEE_ID,
cont>         JH.DEPARTMENT_CODE,
cont>         JH.SUPERVISOR_ID
cont> END_FOR
 EMPLOYEE_ID   DEPARTMENT_CODE   SUPERVISOR_ID
 00171         ENG               00348
 00471         ENG               00348
```

You can also modify values in a field by including a value expression. Any valid arithmetic operation can be used in your value expression to assign new values to a field or fields. You can also refer to values from other fields within the same relation.

For example, if you define a new field in the SALARY_HISTORY relation called WEEKLY, you can assign values to that field for every current record in the SALARY_HISTORY relation. The current record in the SALARY_HISTORY relation, like the current record in the JOB_HISTORY relation, is identified by the MISSING relational operator. The following example uses the MODIFY statement to modify the new WEEKLY field and replace missing values in the WEEKLY field with new values:

```
START_TRANSACTION READ_WRITE RESERVING
             SALARY_HISTORY FOR SHARED WRITE

FOR SH IN SALARY_HISTORY
  WITH SH.SALARY_END MISSING
    MODIFY SH
       USING
          SH.WEEKLY = (SH.SALARY_AMOUNT/52)
    END_MODIFY
END_FOR
```

The MODIFY statement in the preceding example includes a reference to another field, SALARY_AMOUNT, in the same relation and uses it in an arithmetic operation to compute a weekly salary amount for every current employee in the SALARY_HISTORY relation.

6.2.2 Changing Data in Multiple Relations

You can modify data in two or more relations in a single transaction. Specify the relations in the record stream identified by the RSE in the FOR statement, and use a separate MODIFY statement for each relation whose records are to be changed.

For example, assume that because information was unavailable, records in the JOB_HISTORY and SALARY_HISTORY relations were stored with missing values for job code, department code, and salary amount. When the information becomes known, you can locate these records and supply valid data, changing their field values from MISSING to actual values.

Here are the steps to modify the fields in the JOB_HISTORY and SALARY_HISTORY relations:

1. Form an RSE that selects employee records from the EMPLOYEES relation, records for current salaries from the SALARY_HISTORY relation, and records for current jobs from the JOB_HISTORY relation.
2. Modify the field or fields of the record from the SALARY_HISTORY relation.
3. Modify the fields of the record from the JOB_HISTORY relation.
4. Commit the changes to the database.

The following example modifies records in the JOB_HISTORY and SALARY_HISTORY relations, giving each new employee (that is, with an employee ID of 00502 or higher) a job code of MENG, a department code of ENG, and a salary of $25,000:

```
START_TRANSACTION READ_WRITE RESERVING
      EMPLOYEES FOR SHARED READ,
      JOB_HISTORY FOR SHARED WRITE,      ! Start transaction with
      SALARY_HISTORY FOR SHARED WRITE    ! write access

FOR E IN EMPLOYEES WITH                  ! Begin outer loop
     E.EMPLOYEE_ID GE "00502"            ! selecting new employees

 FOR JH IN JOB_HISTORY                   ! Begin first inner loop
   WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID   ! selecting their current
   AND JH.JOB_END MISSING                ! job history records

      MODIFY JH USING                    ! Modify two fields
          JH.JOB_CODE = "MENG";
          JH.DEPARTMENT_CODE = "ENG ";
      END_MODIFY
 END_FOR                                 ! Terminate first inner loop

 FOR SH IN SALARY_HISTORY                ! Begin second inner loop
    WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID  ! selecting their current
    AND SH.SALARY_END MISSING            ! salary history records
```

```
      MODIFY SH USING                    ! Modify salary amount
          SH.SALARY_AMOUNT = 25000
      END_MODIFY

 END_FOR                                 ! Terminate second inner loop
END_FOR                                  ! Terminate outer loop

COMMIT
```

Do *not* attempt to modify any field used in the RSE. If you attempt to modify a join term in the RSE, Rdb/VMS cannot ensure predictable results. Changing such a value might change the contents of the record stream. Note that in the preceding example none of the fields in the RSE itself is modified.

Some update tasks require features, such as extensive arithmetic functions and character manipulation, that a host language program can provide. RDO performs limited queries of the database. It helps you to build logically and syntactically correct statements that you can then embed in a host language program. For extensive and complex interactive updates or reports, you can consider using DATATRIEVE, DECdecision, TEAMDATA, or a RALLY application.

6.3 Erasing Data Records in a Relation

You can delete (erase) one or many records in a relation identified by the RSE using the ERASE statement. Because you are updating the database, you begin the update transaction with a START_TRANSACTION statement specifying READ_WRITE access. When you terminate the transaction, use the COMMIT statement to make the deletions permanent or enter the ROLLBACK statement to restore the database to the state it was in before you made any modifications.

The following example erases the employee record in the EMPLOYEES relation with the employee ID of 00502:

```
START_TRANSACTION READ_WRITE RESERVING
         EMPLOYEES FOR SHARED WRITE

FOR E IN EMPLOYEES
    WITH E.EMPLOYEE_ID = "00502"
      ERASE E
END_FOR

COMMIT
```

To verify that the record for the employee whose identification number is 00502 no longer exists in the EMPLOYEES relation, you can try the next query.

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00502"
     PRINT
       E.EMPLOYEE_ID,
       E.FIRST_NAME,
       E.LAST_NAME
END_FOR
```

RDO does not display any data, indicating that the record with the employee ID of 00502 has been deleted from the EMPLOYEES relation.

In the sample personnel database, deleting the record for employee ID 00502 from the EMPLOYEES relation causes a corresponding deletion of records with that employee ID from the JOB_HISTORY, SALARY_HISTORY, DEGREES, and RESUMES relations. These automatic deletions occur because a *trigger* has been defined specifying those actions. (Section 6.6 discusses triggers.) However, if the trigger had not been defined, there still might be records for employee ID 00502 in the JOB_HISTORY and SALARY_HISTORY relations, and perhaps also in other relations. In such a case, you should write similar statements to delete these records from these relations.

Before you delete the record in the EMPLOYEES relation, you can use the RSE to find the records that will be deleted first (or should be deleted if a trigger has not been defined) in the other relations. After deleting all records associated with employee ID 00502, you can verify that the employee's record has been deleted from the necessary relations by attempting to display records for that employee.

6.4 Updating by Selecting Data from the Record Stream

The DECLARE_STREAM and START_STREAM statements are especially useful for conditionally processing the records in a record stream in programs. The DECLARE_STREAM statement allows you to specify an RSE that will be applied each time the named stream is started. If you use DECLARE_STREAM to specify a record stream, you can use one or more START_STREAM statements that specify just the stream name in the program or session. As an alternative to using DECLARE_STREAM and START_STREAM statements together, you can specify a full RSE with a START_STREAM statement.

Thus, if you plan to use the same stream specification multiple times, it is more convenient to use the DECLARE_STREAM statement once, and then refer to the stream name in each START_STREAM statement. On the other hand, if you plan to use the stream specification only once during the program or session, it is more convenient to give the complete specification with the START_STREAM statement and not use the DECLARE_STREAM statement.

Unlike the FOR statement, which works with every record identified in the RSE until all records are processed, the START_STREAM statement makes records available but does not automatically retrieve any record in a record stream. In order to see the records in a particular record stream, you must retrieve each one with the FETCH statement. After fetching a record, you can test values of fields and process some records while passing over others.

For example, you could update the JOB_HISTORY and SALARY_HISTORY relations with records for new employees, using the following approach. (This example uses the DECLARE_STREAM statement merely for illustration.)

1. Identify and name a group of selected records (for new employees) from the EMPLOYEES relation using the DECLARE_STREAM statement and the EMPLOYEE_ID value for the first new employee.
2. Start the transaction.
3. Locate the record for the first new employee using the FETCH statement.
4. For each new employee record, add a record in the JOB_HISTORY and the SALARY_HISTORY relations.
5. Repeat steps 3 and 4 until there are no more records in the record stream.
6. Close the stream.
7. Commit the transaction to the database.

The following example illustrates the preceding logic except for step 5; that is, the example shows how to add records for the first new employee. (A subsequent example will show how to extend the example to add records for the remaining new employees.)

```
DECLARE_STREAM NEW_STAFF USING E IN EMPLOYEES   !Stream is named NEW_STAFF
        WITH E.EMPLOYEE_ID = "00502"     !ID of first new employee = 00502

START_TRANSACTION READ_WRITE RESERVING
           JOB_HISTORY FOR SHARED WRITE,
           SALARY_HISTORY FOR SHARED WRITE,
           EMPLOYEES FOR SHARED READ

START_STREAM NEW_STAFF

     FETCH NEW_STAFF

       STORE JH IN JOB_HISTORY USING
         JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
         JH.JOB_START = "21-AUG-1989"
       END_STORE

       STORE SH IN SALARY_HISTORY USING
         SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
         SH.SALARY_START = "21-AUG-1989"
       END_STORE

END_STREAM NEW_STAFF

COMMIT
```

The preceding statements add records to the JOB_HISTORY and SALARY_HISTORY relations only for the first new employee. However, assume you have six additional new employee records to process. To solve the problem of processing all six records in the stream interactively, you could repeat the FETCH and STORE statements six times, but this would be very time-consuming. A more efficient method would be to create a command file containing the RDO statements for the operation that you need to perform multiple times (in this case, fetching and storing). For example, use a VMS text editor to create a command file called UPDATE.RDO that contains the following statements:

```
FETCH NEW_STAFF

  STORE JH IN JOB_HISTORY USING
    JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
    JH.JOB_START = "21-AUG-1989"
END_STORE

STORE SH IN SALARY_HISTORY USING
    SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
    SH.SALARY_START = "21-AUG-1989"
END_STORE
```

Then, at the RDO prompt, type the name of the command file preceded by the at sign (@) and have RDO execute the statements in the command file.

Thus, to complete the update operations more efficiently, you can break the process into three steps:

1. Form the record stream with the START_STREAM statement.
2. Execute the command file the required number of times.

3 End the stream and commit the transactions to the database.

The following example finds out how many additional new employees need to be processed, then starts a record stream and invokes the command file UPDATE.RDO the required number of times. (This example does not use the DECLARE_STREAM statement. This approach is taken merely to illustrate the use of the START_STREAM statement with a complete stream specification.)

```
RDO> PRINT COUNT OF E IN EMPLOYEES WITH  !How many more new employees?
cont>      E.EMPLOYEE_ID > "00502"
  6

RDO> START_STREAM NEW_STAFF USING  ! Start stream NEW_STAFF
cont>    E IN EMPLOYEES WITH
cont>    E.EMPLOYEE_ID > "00502"
RDO> @UPDATE                       ! Update - perform 6 times
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> END_STREAM NEW_STAFF          ! Close the stream
RDO> COMMIT                        ! Write changes to database
```

If you had attempted to execute UPDATE.RDO and there were no more records in the record stream, Rdb/VMS would have responded with this message:

```
RDO> @UPDATE
%RDB-E-STREAM_EOF, attempt to fetch past end of stream.
```

Remember to terminate your transactions with either the COMMIT or the ROLLBACK statement. Table 6–1 illustrates the effects of these statements on databases and transactions.

Table 6–1 Effects of COMMIT and ROLLBACK on Databases and Transactions

Items	COMMIT	ROLLBACK
Scope of statement	Includes all invoked databases	Includes all invoked databases
Database	Writes to disk all changes to a database and its data definitions	Does not write to disk any changes to a database and its data definitions
Open streams	Closes all open streams as in END_STREAM	Closes all open streams as in END_STREAM
Position in stream	No record is available	No record is available
Record locks	Releases all locks	Releases all locks

6.5 Using Missing Values

Sometimes you may not have information for every field when you are storing information for a record. When a field in a record is left blank, Rdb/VMS automatically marks the field as missing and sets an internal null flag for that occurrence of the field. You have the option to define a missing value for a field in its field definition. If you do not define a missing value and the field is left blank, Rdb/VMS supplies default missing values to the field in the form of zeros for numeric fields and spaces for text fields. (Do not confuse *missing value* with the SQL *default value*—see Section 6.5.3.)

When a field is defined as missing, Rdb/VMS marks the field and returns the defined missing value when you include the field in display statements. You can think of a missing field as empty. See Section 6.5.1 for more information about retrieving missing field values using the MISSING relational operator. To define a missing value for the field MIDDLE_INITIAL, use the MISSING_ VALUE clause as follows:

```
DEFINE FIELD  MIDDLE_INITIAL
        DESCRIPTION IS /* Employee’s middle initial */
        DATATYPE IS TEXT SIZE IS 1
        MISSING_VALUE IS " ".
```

6.5.1 Retrieving Records with a Missing Field Value

You can retrieve missing field values by using the MISSING relational operator, or by using the PRINT statement with an asterisk (such as with PRINT E.*) to print all the field values including the missing value. When you retrieve a missing field, Rdb/VMS returns the missing value (even though the missing value is not physically stored in the field).

To display the records of any employees who have no middle initial, use the following query:

```
RDO> FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL MISSING
cont> PRINT E.EMPLOYEE_ID, E.FIRST_NAME, E.MIDDLE_INITIAL, E.LAST_NAME
cont> END_FOR
 EMPLOYEE_ID   FIRST_NAME   MIDDLE_INITIAL   LAST_NAME
 00166         Rick                          Dietrich
 00167         Janet                         Kilpatrick
 00168         Norman                        Nash
 00170         Brian                         Wood
 00171         Aruwa                         D’Amico
 . . .
```

Rdb/VMS finds all the records containing fields where no middle initial is assigned and displays the missing value for the MIDDLE_INITIAL field (in this instance, a space) in those records.

You can use the MISSING relational operator to retrieve a segmented string.

When relational operators are used, Rdb/VMS does not include in the record stream fields whose values are missing. Therefore, when using relational operators to make comparisons of fields in the database, be sure to write the RSE clearly to include records whose values are missing.

If a field has a specific missing value defined for it, the missing value is included in the field values displayed. Missing values are assigned the highest value in the ASCII collating sequence. Records sorted by a field with a defined missing value appear first when the sort order is descending and last when the sort order is ascending.

The following items are intended to prevent some common misconceptions about missing values.

- Because a missing value is not actually stored, you cannot retrieve instances of the missing value by specifying the defined missing value in a query. For example, if the missing value for MIDDLE_INITIAL is "" (that is, a single space), the following query will *not* retrieve instances where the MIDDLE_INITIAL field value is missing:

  ```
  FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL = " "
   PRINT E.*
  END_FOR
  ```

  Instead, use:

  ```
  FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL MISSING
   PRINT E.*
  END_FOR
  ```

- If you attempt to store the value defined as the missing value in a field, that value is *not* actually stored; rather, that instance of the field is simply marked as missing. As an illustration, assume that the missing value for the field SEX has been defined as X. As the following example shows, specifying that X be stored in the SEX field for the record with employee ID 00164 does not result in the storage of the value X; the query asking for records WITH E.SEX = "X" returns no records:

  ```
  RDO> START_TRANSACTION READ_WRITE
  RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00164"
  cont> MODIFY E USING E.SEX = "X"
  cont> END_MODIFY
  cont> END_FOR
  RDO> FOR E IN EMPLOYEES WITH E.SEX = "X"
  cont> PRINT E.EMPLOYEE_ID
  cont> END_FOR
  RDO> FOR E IN EMPLOYEES WITH E.SEX MISSING
  cont> PRINT E.EMPLOYEE_ID
  cont> END_FOR
   EMPLOYEE_ID
   00164
  ```

- Similarly, if you use the expression RDB$MISSING in modifying the contents of a field, nothing is physically stored in the field. Rdb/VMS evaluates RDB$MISSING to determine the missing value for a field. As the following example shows, using the expression RDB$MISSING marks the field value as missing; changing the missing value changes what RDO displays when the field value is missing:

```
RDO> START_TRANSACTION READ_WRITE
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00164"
cont> MODIFY E USING E.SEX = RDB$MISSING(E.SEX)
cont> END_MODIFY
cont>END_FOR
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00164"
cont> PRINT E.EMPLOYEE_ID, E.FIRST_NAME, E.LAST_NAME, E.SEX
cont>END_FOR
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME       SEX
 00164         Alvin        Toliver         X
RDO> CHANGE FIELD SEX MISSING_VALUE IS "Z".  ! Change the missing value.
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00164"
cont> PRINT E.EMPLOYEE_ID, E.FIRST_NAME, E.LAST_NAME, E.SEX
cont>END_FOR
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME       SEX
 00164         Alvin        Toliver         Z
```

6.5.1.1 Using Nested FOR Loops, Outer Joins, and the MISSING Clause

You can use nested FOR loops to establish relationships for outer joins. In a common type of join, such as an equijoin, Rdb/VMS matches certain values in a field from one relation with a corresponding field value in another relation. Values that do not match are not included in the join. An outer join also establishes relationships between data items by matching fields, but it includes the unmatched values by adding them to the result of the equijoin.

Note *To allow Rdb/VMS to optimize queries, use nested FOR loops only when you want to reference more than one database or to perform outer joins.*

To accomplish an outer join using Rdb/VMS, you must include an RDB$MISSING clause in the RSE so the unmatched values are added at the end of the join. The RDB$MISSING clause denotes the value of a field has been defined as missing. The following example shows how you use the RDB$MISSING clause in a nested FOR loop to find all employees and show what degrees they have. The employees' last names are sorted in alphabetic order.

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
    FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
        PRINT
           E.LAST_NAME,
           E.FIRST_NAME,
           D.DEGREE,
           D.DEGREE_FIELD

    END_FOR

      FOR FIRST 1 D IN DEGREES
          WITH NOT ANY D1 IN DEGREES
            WITH D1.EMPLOYEE_ID = E.EMPLOYEE_ID

         PRINT
          E.LAST_NAME,
          E.FIRST_NAME,
          RDB$MISSING(D.DEGREE),
          RDB$MISSING(D.DEGREE_FIELD)

    END_FOR

END_FOR
```

This query prints information for all employees. If they have degrees, it prints each degree they have. If an employee has no degrees, the missing value for the degree field is printed (in this case, the value is the word "Unknown"), unless you created the database using the definitions specific to SQL, in which case no missing value is defined (see Section 6.5.3). Because the outer FOR loop sorts the employees by last name, all employees without degrees are included along with the employees who have degrees.

6.5.2 Storing Missing Values

When a field has a defined missing value, you can store a missing value with the STORE statement or you can store a new record and not supply a value for the field you want to have a missing value. The following example shows how to use the STORE statement to store a missing value in the E.MIDDLE_INITIAL field:

```
STORE E IN EMPLOYEES
  USING
     E.MIDDLE_INITIAL = RDB$MISSING(E.MIDDLE_INITIAL)
END_STORE
```

6.5.3 Missing Value Contrasted with SQL Default Value

The missing value for a field is *not* the same as the default value for a column (field) that you can define with the SQL interface. (The SQL statement SHOW TABLE table-name displays the default value for a column as the "Rdb default.") If a store operation does not specify a value for a column with a default value, the default value is actually (physically) stored in the database. If the store operation does not specify a value for a column and the column has no default value defined, then Rdb/VMS stores nothing in that column and sets an internal null flag.

If you use RDO to specify a missing value for a field, then that is the value displayed by RDO when the field has no value stored and the internal null flag is set. SQL, however, does not recognize any missing value specified by RDO; if the column has no value stored and the null flag is set, then SQL displays NULL for the column, regardless of whether you specified any missing value with RDO.

6.6 Referential Integrity and Triggers

The previous sections in this chapter explain how to store, modify, and erase records in a relation. The examples in these sections usually affect records only in the specified relations. However, sometimes decisions made in the design and definition of the database can affect (a) your ability to make changes and (b) the impact of changes you do make. Such decisions include the following:

- A field in a relation can be defined as “referencing” a field in another relation (often the primary key field in the other relation); such a definition establishes a constraint that prevents you from deleting a record that has records in another relation dependent upon it, or from adding or modifying records without a corresponding matching record in another relation.
- A **trigger** causes one or more actions to be performed when a specified type of update operation (deletion, insertion, or modification) is performed.

6.6.1 Using Constraints to Enforce Referential Integrity

Constraints that are established by field references between relations help to preserve the referential integrity of the database, ensuring that no changes are made which would violate certain dependencies among relations. A common use of such constraints is to preserve the integrity of relationships between a primary key and its associated foreign keys.

For example, assume that in the sample personnel database, you wanted to define a constraint by which the EMPLOYEE_ID field in a record in the SALARY_HISTORY relation must match the EMPLOYEE_ID field in a record in the EMPLOYEES relation. There are two ways to define such a constraint. One is to define the constraint separately, as is done in the following example from the file RDM$DEMO:CONSTRAINTS_RDO.RDO:

```
! The employee ID from the SALARY_HISTORY relation must exist in
! the EMPLOYEES relation before it can be stored in the SALARY_HISTORY
! relation.
!
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.
```

Another way to specify the same constraint is to include it in the relation definition. If you have defined the EMPLOYEE_ID field in the EMPLOYEES relation as the primary key, you can use the REFERENCES clause in the definition of the EMPLOYEE_ID field in the SALARY_HISTORY relation to establish the requirement for a match. For example (illustrating only the relevant definitions from the DEFINE RELATION statements):

```
DEFINE RELATION EMPLOYEES.
     EMPLOYEE_ID BASED ON ID_NUMBER PRIMARY KEY.
 . . .
END EMPLOYEES RELATION.

DEFINE RELATION SALARY_HISTORY.
     EMPLOYEE_ID BASED ON ID_NUMBER CONSTRAINT SH_EMP_ID_EXISTS
     REFERENCES EMPLOYEES EMPLOYEE_ID.
 . . .
END SALARY_HISTORY RELATION.
```

Both methods of defining the constraint establish the requirements that any EMPLOYEE_ID field entered in a record in the SALARY_HISTORY relation match an existing EMPLOYEE_ID field in a record in the EMPLOYEES relation, and that no record can be deleted from the EMPLOYEES relation as long as there are any records in the SALARY_HISTORY relation with that person's employee ID. (In the trigger example later in this section, records in the SALARY_HISTORY relation are deleted *before* the associated record in the EMPLOYEES relation is deleted.) Any statement you enter that violates this constraint will fail (at verb time or commit time, depending on when constraint evaluation is performed—see Section 2.3.6).

For further information, see the DEFINE RELATION description in the *VAX Rdb/VMS RDO and RMU Reference Manual*.

6.6.2 Using Triggers

Triggers are often defined to cause one or more actions to be taken automatically when a particular update operation (deletion, insertion, or modification) is performed. The particular operation causes the "triggered action" to take place, affecting fields or even entire records in other relations or in the same relation.

The following example defines a trigger that implements a cascading delete triggered by the deletion of an employee record. The trigger ensures that before an employee is erased from the EMPLOYEES relation, all of his or her records in the DEGREES, JOB_HISTORY, SALARY_HISTORY, and RESUMES relations will also be erased. This trigger also ensures that if the employee in question is also a department manager, the MANAGER_ID field for that department will be marked as missing. (See Section 6.5 for a discussion of missing values.)

```
!
DEFINE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
      BEFORE ERASE
      FOR E IN EMPLOYEES EXECUTE
        FOR D IN DEGREES WITH
          D.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE D
        END_FOR;
        FOR JH IN JOB_HISTORY WITH
          JH.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE JH
        END_FOR;
        FOR R IN RESUMES WITH
          R.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE R
        END_FOR;
        FOR SH IN SALARY_HISTORY WITH
          SH.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE SH
        END_FOR;
!  Also, if an employee is terminated and that employee is
!  the manager of a department, set the MANAGER_ID missing for
!  that department.
        FOR D IN DEPARTMENTS WITH D.MANAGER_ID = E.EMPLOYEE_ID
          MODIFY D USING D.MANAGER_ID = RDB$MISSING (D.MANAGER_ID)
          END_MODIFY
        END_FOR
      FOR EACH RECORD.
```

For more information on triggers, see the DEFINE TRIGGER section in the *VAX Rdb/VMS RDO and RMU Reference Manual*.

Note *Triggers are not necessarily related to the referential integrity of a database; however, triggers are often used conjunction with other features to ensure referential integrity.*

The execution of a trigger action is *not* guaranteed to occur at any specific point within the transaction; the only guarantee is that the cumulative impact of any trigger actions will be in effect when the transaction is committed. Thus, you should not assume that any specific trigger action will be executed immediately after the statement triggering it.

For example, assume that the following trigger has been defined to calculate the next sequence number to be assigned (by adding one to the count of orders):

```
DEFINE TRIGGER SEQUENCE_NUM_TRIG AFTER STORE
     FOR O IN ORDERS_TABLE EXECUTE
     FOR S IN SEQ_TABLE
     MODIFY S USING S.NUMBER = COUNT OF OT IN ORDERS_TABLE + 1
     END_MODIFY
     END_FOR
     END_FOR
   FOR EACH RECORD.
```

Assume that the ORDERS_TABLE relation contains 99 records, and the value of the SEQ_TABLE.NUMBER field is 100. Your application then stores 10 new records in the ORDERS_TABLE relation within a single transaction. Under the current implementation, each record insertion causes the SEQ_ TABLE.NUMBER field's value to be updated; thus, after the 100th ORDERS_ TABLE record is inserted, the NUMBER field is set to 101; after the 101st ORDERS_TABLE record is inserted, the NUMBER field is set to 102; and so forth.

However, this implementation may change in the future, so that the trigger actions are performed at the end, thus causing the value of the SEQ_ TABLE.NUMBER field to increase from 100 to 110 only when the transaction is committed (that is, after all 10 insertions). Therefore, be sure to design applications so they do not depend on a particular timing of trigger actions within a transaction.

6.7 Summary

The following session demonstrates a sequence of RDO statements that update the database. Note that the STORE statements neglect to store all information normally included in the records—this is done merely to simplify the illustration (and your typing work if you are entering these statements), not to depict any real application.

```
!
! Invoke the PERSONNEL database.
!
RDO> INVOKE DATABASE PATHNAME 'PERSONNEL'

!
! Signal your access (update) intentions to Rdb/VMS.
!
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         EMPLOYEES FOR SHARED WRITE


!
! Store a new employee record.
!
RDO> STORE E IN EMPLOYEES USING
cont>   E.EMPLOYEE_ID = "00503";
cont>   E.FIRST_NAME = "Paul";
cont>   E.LAST_NAME = "Cranston"
cont> END_STORE

!
! Make the update permanent.
!
RDO> COMMIT
```

```
!
! Store a new salary_history record.
!
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         SALARY_HISTORY FOR SHARED WRITE,

RDO> STORE SH IN SALARY_HISTORY USING
cont>         SH.EMPLOYEE_ID = "00503"
cont>END_STORE

!
! Make the update permanent.
!
RDO> COMMIT

!
! Start a read/write transaction so you can modify and erase
! data; reserve the necessary relations.
!
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         EMPLOYEES FOR SHARED WRITE,
cont>         SALARY_HISTORY FOR SHARED WRITE

!
! Make the first change.
!
RDO>  FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>   CROSS SH.SALARY_HISTORY OVER EMPLOYEE_ID
cont>   WITH SH.SALARY_END MISSING
cont>     MODIFY SH USING
cont>       SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * 1.9
cont>     END_MODIFY
cont> END_FOR

!
! Mistake! Percent for raise is incorrect; undo the change.
!
RDO> ROLLBACK

!
! Start the transaction again.
!
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         EMPLOYEES FOR SHARED WRITE,
cont>         SALARY_HISTORY FOR SHARED WRITE

!
! Specify the correct percent for the raise.
!
RDO>  FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>   CROSS SALARY_HISTORY OVER EMPLOYEE_ID
cont>   WITH SH.SALARY_END MISSING
cont>     MODIFY SH USING
cont>       SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * 1.1
cont>     END_MODIFY
cont> END_FOR
```

```
!
! Erase employee and salary_history records with
! employee ID 00503.
!
RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>   ERASE E
cont> END_FOR
RDO> FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = "00503"
cont>   ERASE SH
cont> END_FOR

!
! Make the change and the deletion permanent.
!
RDO> COMMIT
RDO>
```

7

Introduction to Rdb/VMS Programming

This chapter introduces Rdb/VMS programming concepts, including a description of the programming interfaces and how to develop a program prototype using RDO.

7.1 The Programming Interfaces

Rdb/VMS provides the following programming interfaces that let you access one or more Rdb/VMS databases:

- The RDBPRE preprocessor

  This preprocessor lets you access an Rdb/VMS database from BASIC, COBOL, or FORTRAN programs. The RDBPRE preprocessor lets you embed Rdb/VMS statements directly in your host language program. However, you must use the Callable RDO interface to perform data definition tasks.

- The RDML preprocessor

  This preprocessor lets you access an Rdb/VMS database from C or Pascal programs. The RDML preprocessor lets you embed RDML statements directly in your host language program. However, RDML also requires that you use the Callable RDO interface to perform data definition tasks.

- The Callable RDO interface

  This interface lets you access an Rdb/VMS database from any host language supported by the VAX Procedure Calling Standard. You may also use this interface, as mentioned previously, when you want to perform Rdb/VMS data definition tasks from a program.

- The SQL precompiler

  This interface lets you access an Rdb/VMS database from Ada, C, COBOL, FORTRAN, Pascal, or PL/I programs. The SQL precompiler allows you to embed SQL statements directly in host language modules.

- The SQL module processor

  This processor lets you link one or more SQL modules with one or more host language modules. Statements in an SQL language module are in the form of uniquely named procedures that you call from a host language module.

Because RDBPRE and RDML support neither Ada nor PL/I, you may want to investigate the possibility of using SQL if you want to program in these languages. Although you can use the Callable RDO interface, your program will be more efficient if you use the SQL precompiler or the SQL module processor with these languages.

For more information on the SQL precompiler and the SQL module processor refer to the *VAX Rdb/VMS Guide to Using SQL*.

RDBPRE and RDML check the syntax of data manipulation language (DML) statements you use in your program. Reference material, such as syntax diagrams, can be found in the Rdb/VMS reference manuals. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for RDBPRE reference material. The *RDML Reference Manual* contains reference material for RDML.

The Callable RDO program interface, RDB$INTERPRET, accepts Rdb/VMS statements as strings. When your program executes, these statements are passed to Rdb/VMS in calls to the RDB$INTERPRET function. The interactive interface, RDO, then interprets and executes them. The *VAX Rdb/VMS RDO and RMU Reference Manual* contains reference material for the RDO utility.

All VAX languages that support the VAX Procedure Calling Standard can use the Callable RDO program interface. You must use this interface when Rdb/VMS does not support a preprocessor for your program language or when you want to perform Rdb/VMS data definition tasks in RDML or RDBPRE programs.

Note that the Callable RDO program interface uses significantly more resources than either RDBPRE or RDML. For this reason, if possible, you should use RDBPRE or RDML when you are programming in BASIC, C, COBOL, FORTRAN, or Pascal. However, if you need to perform data definition tasks within these programs, keep in mind that you can use RDML or RDBPRE statements and the Callable RDO program interface within the same program.

7.2 Designing a Prototype Using RDO

Before you write your application program, you need to familiarize yourself with the database that your program will access and determine how you will form the database queries you intend to include in your program. Designing a transaction prototype using RDO can simplify the task of writing the application program. Debugging your program is likely to be a tedious task if you have not previously created a prototype for your queries.

7.3 Developing Your Queries with an Interactive Interface

There are two interactive interfaces for the development of Rdb/VMS queries:

- The Relational Database Operator (RDO) utility
- Interactive SQL

For information on interactive SQL, refer to the *VAX Rdb/VMS Guide to Using SQL*. The rest of this chapter discusses developing a prototype using the RDO interface. RDO lets you:

- View database characteristics

  You can use the RDO SHOW statement to display information about the database. The RDO SHOW statement is discussed in the next section.

- Detect and correct errors in Rdb/VMS syntax

  An interactive interface is the best environment to eliminate syntax errors in Rdb/VMS statements. You can use the RDO EDIT statement to correct a problem when RDO indicates an error. Although the syntax for both RDBPRE and RDML differs slightly from the RDO syntax for some statements, you should still be able to eliminate most of the problem areas from your program design before you actually code your application program. You can use the RDO SET OUTPUT statement (described later in this chapter) to write your interactive RDO session to a log file. You can then review the errors you made and how you corrected them when you write your application program.

- Evaluate the effectiveness of your queries

  Manipulating data at the relation level is very different from manipulating data at the record level. You can use RDO to determine efficient forms for a query. In general, you do not want to spend a lot of time optimizing every query in your program. If you have a query that seems to run particularly slowly, it is a good idea to investigate alternative forms for the query to find a more efficient way of accessing the database.

- Examine the types of data that your program must handle

  You can determine the data types your program must handle and test data input and output values so that your program handles database values intelligently. You can also determine if your program should enforce its own input validity checks in addition to the checks that may be provided by the database.

- Anticipate Rdb/VMS run-time errors

  Using RDO gives you a clearer idea of the run-time errors your program may encounter. You can use RDO to test the conditions that produce a particular run-time error.

7.3.1 Using the RDO SHOW Statements

You can use the RDO SHOW statements to familiarize yourself with the database. Some of the database characteristics that you might want to see include:

- Relation and view definitions

  Knowing what relations and views the database contains will have an impact on the types of queries you can make. You should consider the type of information your application will need to retrieve from the database and how the relations and views can be joined to retrieve that information.

- Field data types and sizes

  Your program usually needs to declare host language variables that pass values to, and accept values from, the database. You need to be aware of the data types defined for the fields you will be using, and which of these data types is compatible with the host language you are using. Refer to Chapter 8 for tables that show Rdb/VMS data types and compatible data types in BASIC, C, COBOL, FORTRAN, and Pascal.

- Data validation checks

  These checks can be built into the database by the database designer. They may restrict the values that may be stored in a relation. Your application design may want to take these restrictions into account so you can direct your users not to attempt operations that would store the restricted data, or so your program can handle situations when the user attempts to store these values.

- Index design and definitions

  By paying attention to how indexes have been designed and defined for your database, you can have a considerable impact on the efficiency of your program. Taking advantage of an index can mean the difference between a program that processes quickly and one that processes slowly. Refer to the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for more information on how indexes affect query performance.

- Storage map design and definitions

  If your Rdb/VMS database is a *multifile database*, one that is spread over multiple physical areas in separate VMS files, you should pay attention to how relations and indexes are distributed across files by studying storage map design and definition.

  A storage map specifies how a relation is mapped to a storage area. It tells Rdb/VMS in which area or areas a relation can be stored, how a relation that is to be spread across multiple storage area files will be distributed, what storage method Rdb/VMS should use to determine the location of a record initially being stored in a relation, and whether or not records in a relation will be compressed. By studying the storage maps defined for your database, you may be able to form queries that take advantage of the database structure most efficiently.

  See the *VAX Rdb/VMS Guide to Database Design and Definition* for information on storage maps.

7.3.2 Determining Which Relations and Views to Use

As you develop a prototype of your program, you will need to consider how you will retrieve information from the database. Part of this determination will depend on how relations and views have been defined in the database. If you do not effectively use the database attributes available to you, you could end up with an inefficient query that looks like the following:

```
FOR E IN EMPLOYEES CROSS JH IN JOB_HISTORY
  CROSS SE IN EMPLOYEES CROSS SJH IN JOB_HISTORY
    WITH JH.JOB_END MISSING
      AND JH.EMPLOYEE_ID = E.EMPLOYEE_ID
      AND E.EMPLOYEE_ID = '00205'
      AND SJH.JOB_END MISSING
      AND SJH.EMPLOYEE_ID = SE.EMPLOYEE_ID
      AND SE.EMPLOYEE_ID = JH.SUPERVISOR_ID
          .
          .
          .
END_FOR
```

Rdb/VMS databases store view definitions in binary language representation (BLR). The BLR code is an efficient representation of a query. This last query could be formed more efficiently by using the view, CURRENT_JOB. For example:

```
FOR CJ IN CURRENT_JOB
  CROSS CJ2 IN CURRENT_JOB
    WITH CJ.EMPLOYEE_ID = '00205'
    AND CJ2.EMPLOYEE_ID = CJ.SUPERVISOR_ID
        .
        .
        .
END_FOR
```

7.3.3 Determining Data Types of Database Fields

Your program variables must match their corresponding fields in the database. The interactive RDO interface gives you the ability to determine:

- Field names, data types, and sizes
- Indexed fields
- Valid input values for database updates

You can use the RDO SHOW statement to display database attributes on your terminal. The SHOW statement is especially useful when you want to copy data dictionary definitions for database relations and fields into your program. Use the SHOW statement to determine which data dictionary definitions:

- You can copy without modification
- Contain data types your programming language does not support
- Contain non-unique field names

The following example stores in the file SHOWDB.TXT the names of all relations and field definitions for the EMPLOYEES, JOB_HISTORY, and SALARY_HISTORY relations from the PERSONNEL database. Note that you must first invoke a database before you can use the SHOW statement for that database.

```
RDO> SET OUTPUT SHOWDB.TXT
RDO> INVOKE DATABASE FILENAME 'MF_PERSONNEL'
RDO> SHOW RELATIONS
RDO> SHOW FIELDS FOR RELATION EMPLOYEES,JOB_HISTORY,SALARY_HISTORY
RDO> SET NOOUTPUT
```

For additional information on using the SHOW statement, see the *VAX Rdb/VMS RDO and RMU Reference Manual*.

7.3.3.1 Determining Data Validation Checks Defined for the Database

As you develop your prototype, you will want to keep in mind the various ways that Rdb/VMS performs data validation. Rdb/VMS can check validity and enforce constraints on input values by checking for:

- Constraint violations

  Your database designer may use the DEFINE CONSTRAINT statement to set conditions that restrict the values stored in a relation. For example, a constraint could require that a department code must exist in the DEPARTMENTS relation before a record with that department code can be stored in the JOB_HISTORY relation.

- Violation of the VALID IF clause

  Your database designer may use the VALID IF clause in the DEFINE FIELD or CHANGE FIELD statements to set conditions that restrict the values stored in a given field. For example, a VALID IF clause can require that an employee ID lie within a certain range of values.

- Violation of the DUPLICATES ARE NOT ALLOWED clause

  Your database designer may use the DUPLICATES ARE NOT ALLOWED clause in the DEFINE INDEX statement to require that each value in the index be unique. If this is the case, you must be certain that your application either forces the user to choose a unique value for the field on which the index is defined, or, handles the error should the user enter a non-unique value.

If your database uses these validity checks, your program does not need to check for valid input data. However, your program must detect the error condition that Rdb/VMS returns when an integrity failure or constraint violation occurs. Your application determines how to handle the error once it is identified. (See Chapter 10 for more information on error handling.)

The DEFINE CONSTRAINT statements are checked at the same level at which they are defined. Therefore, if you define constraints at the record level, and your input data involves several fields that could violate the same constraint, you will not know which field is invalid if you have a constraint violation.

If you choose to design your input record so that no constraint can be violated by more than one field, you may increase the overhead associated with checking constraints. For example, if a relation defines constraints such that two field values in one relation cannot exist until those two field values exist in another relation, Rdb/VMS must do twice as much work to check those field constraints to make sure that a record exists in one relation before it can exist in another relation.

Alternatively, you can design your program logic to check the validity of data before you attempt the store operation.

Another aspect of data validation is to ensure the consistency of the data in the database. One Rdb/VMS feature that can be used to help ensure consistency is a trigger. Your database designer may use the DEFINE TRIGGER statement to create a trigger for a relation. For example, a trigger could require that when you delete an employee record from the EMPLOYEES relation, employee records from other relations that have foreign keys referring to the primary key in the EMPLOYEES relation are also deleted. For more information, see the DEFINE TRIGGER statement in the *VAX Rdb/VMS RDO and RMU Reference Manual*.

7.3.3.2 Using the RDO SET OUTPUT Statement The SET OUTPUT statement creates an output file and writes an entire RDO session into this file until the command is turned off with the SET NOOUTPUT statement. Your queries, the data returned by Rdb/VMS, syntax error messages for invalid RDO statements and your EDIT statements, are all included in the output file. You can store your prototype queries in this file, and then call the file into an editing buffer when you create your host language program. (See Section 7.3.3.3, Statement Testing in RDO.) The following example opens an output file called QUERY.LOG with the SET OUTPUT statement. It stores the RDO session in QUERY.LOG until the file is closed by the SET NOOUTPUT statement.

```
RDO>   !Open an output file.
RDO> SET OUTPUT QUERY.LOG
RDO> INVOKE DATABASE FILENAME 'PERSONNEL'
RDO> FOR D IN DEPARTMENTS CROSS E IN EMPLOYEES WITH
cont> D.MANAGER_ID = E.EMPLOYEE_ID AND D.DEPARTMENT_CODE = 'PUBL'
cont> PRINT D.MANAGER_ID, E.LAST_NAME END_FOR
RDO>   !Department PUBL does not exist; RDO returns no data.
RDO>
RDO>   !End read-only transaction, start read/write transaction.
RDO> COMMIT
RDO> START_TRANSACTION READ_WRITE RESERVING DEPARTMENTS FOR
cont> EXCLUSIVE WRITE, EMPLOYEES FOR SHARED WRITE,
cont> JOB_HISTORY FOR SHARED WRITE
RDO>   !Store the department.
RDO> STORE D IN DEPARTMENTS USING
cont> D.DEPARTMENT_CODE = 'PUBL';
cont> D.DEPARTMENT_NAME = 'Publicity';
cont> D.MANAGER_ID = '00225' END_STORE
RDO>   !see if department is stored
RDO> FOR D IN DEPARTMENTS WITH D.DEPARTMENT_CODE =
cont> 'PUBL' PRINT D.DEPARTMENT_NAME END_FOR
RDO>
   D.DEPARTMENT_NAME
   'Publicity'
RDO>   !Now close the output file.
RDO> SET NOOUTPUT
```

For additional information about using the SET OUTPUT statement, see Section 3.2.

7.3.3.3 Statement Testing in RDO

The interactive RDO utility gives you the opportunity to test most of your Rdb/VMS data manipulation statements before coding and running your program. After this testing, you can be reasonably assured that the Rdb/VMS statements in your program form the required record streams and retrieve and update the required records.

Although you should test as many of your data manipulation statements as possible, it is particularly important that you test those statements that form record streams. These record streams form the sources for data retrieval and the targets for updates. The RSE is the key element of a statement that forms a record stream.

You should be aware of the following differences between RDO, and RDBPRE and RDML statements when you test statements in interactive RDO:

- In RDBPRE, RDML, and Callable RDO programs, you can use the GET statement to retrieve records from a record stream. In RDML programs, any reference to a field name will retrieve the records from a record stream; for example, a host language print statement or an assignment statement will return a field value to your program. In interactive RDO, you must use the RDO PRINT statement.
- In RDML, RDBPRE and Callable RDO programs, you use the START_STREAM and FETCH statements to exercise complete control over the retrieval of records from a record stream. You can nest these statements within a host language loop that controls the iteration of the FETCH retrieval operation. You can also spread these statements across routines in a single module.

  In interactive RDO there is no mechanism to create a loop, only the FOR statement is iterative. When you want to test a START_STREAM . . . FETCH . . . PRINT operation, you must explicitly repeat the FETCH and PRINT statements within the stream context.
- In RDML and RDBPRE programs, you use your host language concatenation operator (if your host language supports one) to concatenate Rdb/VMS fields you retrieve into your program.
- In RDML and RDBPRE programs you can use the concatenation operator (|) to concatenate values only within an RSE.

  In interactive RDO you can use the concatenation operator (|) in all appropriate situations (not just within an RSE).
- You can only use the ON ERROR clause in preprocessed programs. You cannot use the ON ERROR clause in interactive RDO or Callable RDO. Be sure to test the ON ERROR clauses when you debug your preprocessed program.

You can create an RDO prototype using either of the following methods:

- Creating an RDO test file

  The recommended method is to use a text editor, such as VAX EDT or VAX Text Processing Utility (VAXTPU), to enter the statements you want to test into a file. After you have edited the required Rdb/VMS statements, close the file and start an RDO session. To execute your test file, type the at sign (@) immediately followed by the file name. For example, if your file is named TEST.RDO, type @TEST.RDO at the RDO> prompt to run the file. RDO executes the statements in the order they appear in the file. (If you omit the file type, the default file type RDO is used.)

  The first error causes RDO to stop executing the statements and return an error message. Type EDIT 0 to invoke EDT within RDO. (Or you can invoke VAXTPU if you prefer.) Include your test file in the edit buffer and correct the invalid statement. Insert the ROLLBACK statement at the beginning of the edit buffer, so that this new run starts fresh. When you exit the editor with the EXIT command, RDO executes your test file.

  Repeat this process until the test file gives you a clean run and yields the desired results. When you edit your source program, include the test file in your program file and make the necessary adaptations to the program environment.

- Creating an RDO log file

  Start an RDO session and create a log file of the session by issuing the SET OUTPUT statement with a file name. Sequentially enter and execute all of your statements. Whenever RDO returns an error message, type EDIT *n* to invoke the RDO editor and correct the invalid statement.

  The value you use for *n* depends on how many of the previously executed statements you want to display in the editor. You can enter, for example, EDIT, EDIT 2, or EDIT * to edit the last statement, the last two statements, or all the statements in the current RDO session. You can use the RDO statement SET EDIT KEEP *n* to specify the number of statements RDO includes in the editor when you type EDIT * (by default this number is 20).

  When you exit the editor with the EXIT command, RDO continues execution, beginning with the first statement in the edit buffer. Repeat this process until you get a clean run that yields the desired results. Then exit RDO and edit the log file by deleting all but the final set of statements that gave a clean run. Now you can include this file in your source program file and make the necessary adaptations to the program environment.

The following example is a simple prototype of a query in RDO. This prototype merely stores a record in the CANDIDATES relation. Example 13–1, Example 14–1, Example 15–1, Example 17–1, and Example 18–1 show how you might code this query in your host language of BASIC, COBOL, FORTRAN, C, or Pascal, respectively.

```
! Set verify.
!
! Store new candidate relation.
!
!
START_TRANSACTION READ_WRITE RESERVING
        CANDIDATES FOR SHARED WRITE

     STORE C IN CANDIDATES USING
        C.LAST_NAME = Stewart;
        C.FIRST_NAME = Allyn;
        C.MIDDLE_INITIAL = I;
        C.CANDIDATE_STATUS = "Available July 1, 1989";
     END_STORE

!
!Repeat STORE statement for any additional candidates.
!
COMMIT
```

Note *If you are using an active database when you test and you are testing any of the update statements (STORE, MODIFY, or ERASE), be sure to end all transactions with a ROLLBACK statement. Failure to roll back will change the values in the database permanently.*

The language-specific chapters contain examples of converting an RDO prototype to a host language program.

8 Data Type Compatibility

This chapter describes how to select host language data types that are compatible with Rdb/VMS data types. Rdb/VMS supports nine VMS data types, and a special Rdb/VMS data type. These data types are:

- SIGNED BYTE
- SIGNED WORD
- SIGNED LONGWORD
- SIGNED QUADWORD
- F_FLOATING
- G_FLOATING
- DATE
- TEXT
- VARYING STRING

The special data type is SEGMENTED STRING.

This chapter discusses the nine data types and the Rdb/VMS segmented string, and which ones are acceptable to use in host language programs that access an Rdb/VMS database.

For information on the methods you can use to declare these data types in an RDBPRE or RDML program, refer to Chapter 12 and Chapter 16.

8.1 Rdb/VMS Data Types

When you choose a data type for a host language variable that receives or sends data to an Rdb/VMS database, it should match the Rdb/VMS data type wherever possible. In some instances, however, you may be able to use host language data types that Rdb/VMS does not support.

For example, because Rdb/VMS does not store data in the form of the PACKED DECIMAL data type, it is not considered one of the Rdb/VMS data types. However, Rdb/VMS can accept PACKED DECIMAL data from a BASIC or COBOL variable, and can return data to a PACKED DECIMAL variable in a host language program. Thus, you can use this particular data type in programs that access an Rdb/VMS database (although it incurs additional overhead).

In some instances, the host language preprocessor may attempt to assign a host language data type that is compatible with the data type declared by the Rdb/VMS preprocessor. Whether or not your host programming language performs such data conversions depends on the flexibility of the individual language. Refer to your host language user's guide for additional information on data type conversions.

Table 8–1 is a summary of the data types supported by Rdb/VMS and the comparable VMS data types.

Table 8–1 Rdb/VMS Data Types

Rdb/VMS Data Type	Comparable VMS Data Type	VMS Name	Size	Range/Precision
SIGNED BYTE	Signed byte integer	DSC$K_DTYPE_B	8 bits	integer range of –128 to 127
SIGNED WORD	Signed word integer	DSC$K_DTYPE_W	16 bits	integer range of –32768 to 32767
SIGNED LONGWORD	Signed longword integer	DSC$K_DTYPE_L	32 bits	integer range of –2**31 to (2**31)–1
SIGNED QUADWORD	Signed quadword integer	DSC$K_DTYPE_Q	64 bits	integer range of –2**63 to (2**63)–1
F_FLOATING	F_FLOATING Single precision, floating-point number	DSC$K_DTYPE_F	32 bits	0.29 x 10**(–38) to 1.7 x 10**38 Approximately 7 decimal digits

(continued on next page)

Table 8–1 (Cont.) Rdb/VMS Data Types

Rdb/VMS Data Type	Comparable VMS Data Type	VMS Name	Size	Range/Precision
G_FLOATING	G_FLOATING Extended precision, floating-point number	DSC$K_DTYPE_G	64 bits	0.56 x 10**(–308) to 0.9 x 10**308 Approximately 15 decimal digits
DATE	Absolute date and time	DSC$K_DTYPE_ADT	64 bits	Not applicable
TEXT	ASCII text	DSC$K_DTYPE_T	n bytes[1]	0 to 16383 characters
VARYING_ STRING	ASCII text	DSC$K_DTYPE_VT	n + 2 bytes[2]	0 to 32767 characters
SEGMENTED STRING	None	Rdb/VMS specific	Varies	0 to 64K bytes per segment

[1]The "n" is an unsigned integer that represents the number of characters.

[2]The extra 2 bytes added for the VARYING STRING data type is a word used to hold the count of characters in the varying string.

8.2 The Segmented String Data Type

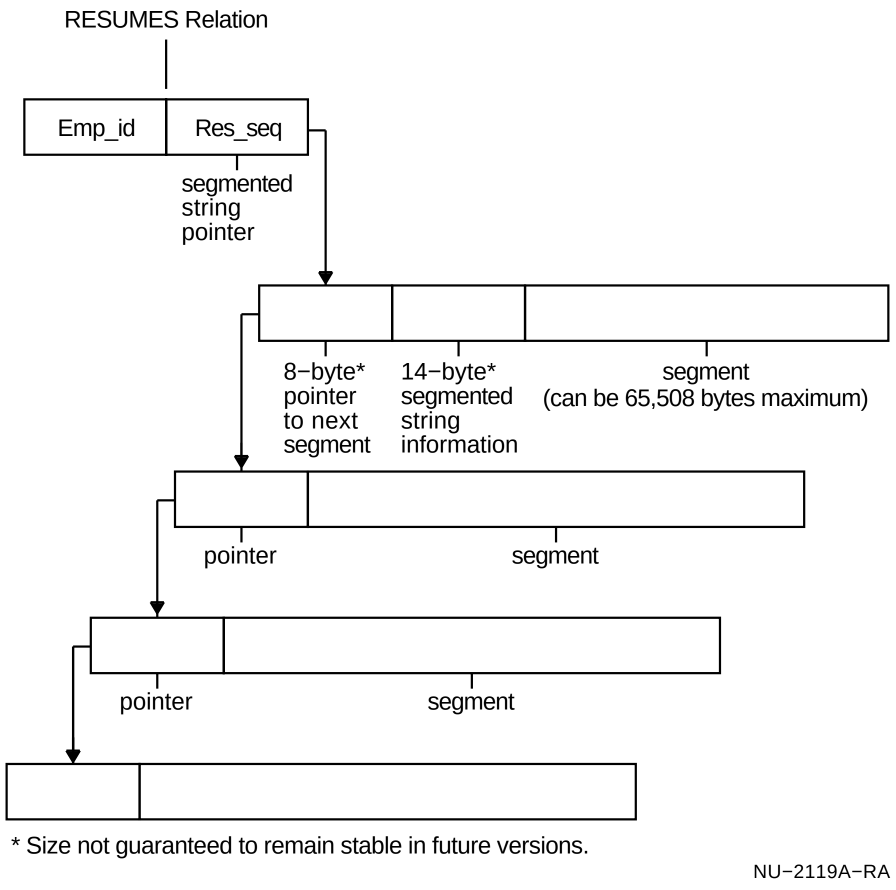
You can use the Rdb/VMS SEGMENTED STRING data type to store large blocks of data in a database. The SEGMENTED STRING data type lets you store unstructured data such as text, graphics, voice, telemetry, or bit streams. Any data type can be stored in and retrieved from a segmented string. The data is stored in unstructured bytes. For example, you can store character data into a segmented string and then interpret it as hexadecimal data.

A segmented string data type is a linked list of vectors (one or more segments that comprise the segmented string). (A **vector** is a one-dimensional array.) Each segment can be up to 65,522 bytes long, except for the first segment of the string, which has a maximum length of 65,508 bytes. The first segment uses an additional 14 bytes for the overhead involved in maintaining a segmented string. The first 8 bytes of each segment is a pointer to the next segment.

The Rdb/VMS preprocessors require that you supply a static class string descriptor when you pass segmented strings between the database and your host language variables. You cannot use a dynamic class descriptor.

A segmented string is stored in a field in a relation. In fact, you actually store a segmented string identifier in the field with the segmented string data type. Because you store a pointer to the segmented string record, rather than the string itself, the segmented string is not constrained by the Rdb/VMS record size limit. Note that because the segmented string itself is not actually stored in a field of the record, you cannot use the RDML data declaration statement, DECLARE_VARIABLE, to declare a variable to hold a segmented string. See Chapter 16 for details. Figure 8–1 illustrates the structure of the segmented string data type.

Figure 8–1 The Segmented String Data Type


RESUMES Relation
Emp_id
Res_seq
segmented string pointer
8−byte* pointer to next segment
14−byte* segmented string information
segment (can be 65,508 bytes maximum)
pointer
segment
pointer
segment
* Size not guaranteed to remain stable in future versions.
NU−2119A−RA

8.3 Data Type Conversions

When your host language program accesses an Rdb/VMS data type that is not supported by your host language, or requests the result of a statistical expression, Rdb/VMS attempts to perform data type conversions where possible, and then pass the database values to your host language variables. Rdb/VMS converts data types for:

- RDBPRE programs
- RDML programs
- Callable RDO programs
- Statistical expressions

The segmented string data type has no corresponding VMS data type. For that reason, Rdb/VMS does not convert the segmented string data type. Instead, your program must explicitly process each segment of the segmented string. See the language-specific chapters for an explanation of how to access and manipulate a segmented string in your programming language.

8.3.1 Preprocessed Program Data Type Conversions

The RDML and RDBPRE preprocessors declare variables that act as intermediaries between your host language variables and the database values. The data type assigned to an intermediate variable depends on the data type of the database field you access and the preprocessor you use.

When a host language data type is the same as the data type of the database value, the preprocessor declares a variable of that data type and no data type conversion takes place. However, when your host language does not support the Rdb/VMS data type, the preprocessor declares an intermediate variable that is supported by the host language. Rdb/VMS converts the database value to or from this intermediate data type when it passes the value to or receives the value from the host language program. When you declare your host language variables, choose the same or an equivalent data type to the Rdb/VMS data type. In RDML, the DECLARE_VARIABLE clause and the BASED_ON clause will make these declarations for you. For more information on using the DECLARE_VARIABLE and BASED_ON clauses see Chapter 16.

Table 8–4, Table 8–5, Table 8–6, Table 8–7, and Table 8–8 list the Rdb/VMS data types you should use to pass values to and from your host language program. Note that the data dictionary, CDD/Plus, does not create an acceptable data type for those data types that are marked with a dagger (†). These data types require that you perform an appropriate data type conversion or manipulation in your host language program.

For example, neither RDBPRE nor RDML converts segmented strings. Instead, you must construct and manipulate fields of the SEGMENTED STRING data type within your program.

8.3.2 Callable RDO Program Data Type Conversions

If you are embedding Callable RDO in a language supported by one of the preprocessors, use the table associated with that language to select host language variable data types for data declarations. If you are embedding Callable RDO in a language that is not supported by a preprocessor, you need to work out acceptable data types by referring to Table 8–1 and your host language documentation. You may find it helpful to generate a table similar to the ones shown here for whatever language you are using.

Callable RDO programs access RDO through the function RDB$INTERPRET. You pass the RDO command string and host language variables to the database as parameters of RDB$INTERPRET. Keep in mind that the RDB$INTERPRET function requires all parameters to be passed by descriptor.

8.3.3 Statistical Expression Data Type Conversions

When passing the result of a statistical expression, RDBPRE or RDML may assign a data type to the result that is different from the data type of the field referred to in the expression. If the result data type is not supported by your host language, RDBPRE or RDML performs the data conversions listed in Table 8–2 and Table 8–3.

Table 8–2 RDBPRE and RDO Statistical Expression Data Type Conversions

Statistical Function	Field Data Type	Result Data Type
MIN, MAX	Any	Same as field
COUNT	Any	LONGWORD
AVERAGE	WORD, F_FLOATING	F_FLOATING (G_FLOATING for larger fields)
TOTAL	F_FLOATING, G_FLOATING Other numeric	QUADWORD

Table 8–3 RDML Statistical Expression Data Type Conversions

Statistical Function	Field Data Type	Result Data Type	C Equivalent	Pascal Equivalent	EPascal Equivalent
MIN, MAX	Any	Same as field	Same as field	Same as field	Same as field
COUNT	Any	Longword	int, long	INTEGER	INTEGER
AVERAGE	Any	F_FLOATING	float	SINGLE, REAL	REAL
TOTAL	Any	G_FLOATING	double	DOUBLE	DOUBLE

8.4 Host Language Equivalent Data Types

Table 8–4, Table 8–5, Table 8–6, Table 8–7, and Table 8–8 list the C, BASIC, COBOL, FORTRAN, and Pascal data types that can be used to declare variables to hold database field values in RDBPRE and RDML programs.

Note that the Rdb/VMS DATE data type is in the 64-bit VMS system time format. Rdb/VMS stores the DATE data type in:

- An 8-byte character data in BASIC, C, and FORTRAN
- An 8-byte record in Pascal
- An 8-byte computational data in COBOL

You can use the VMS system service routine SYS$ASCTIM to convert this 8-byte data into an ASCII string when you want to display a DATE data type field. Use the VMS system service routine SYS$BINTIM to convert an ASCII string into the 64-bit system time format when you want to store a DATE data type field. See the host language chapters for examples of using these system services.

RDBPRE and RDML let you store the DATE data type in either an Rdb/VMS DATE data type field or an Rdb/VMS TEXT data type field. You can, for instance, change an Rdb/VMS DATE field to TEXT without affecting the DATE data type records stored in that field. If you change the data type of the field, you must preprocess the programs that use that field again. Furthermore, when you retrieve a DATE data type field from a TEXT field, the text string retrieved will be in the form: yyyymmddhhmmsshh (year, month, day, hour, minutes, seconds, hundredths of a second).

Table 8–4 Rdb/VMS Data Type Conversions for BASIC

If the Rdb/VMS Data Type Is:	Declare Your BASIC Variable as:
SIGNED BYTE	BYTE
SIGNED BYTE SCALE n†[1]	DECIMAL or any data type
SIGNED BYTE SCALE –n†[2]	DECIMAL or any data type
SIGNED WORD	WORD
SIGNED WORD SCALE n†[1]	DECIMAL or any data type
SIGNED WORD SCALE –n†[2]	DECIMAL or any data type
SIGNED LONGWORD	LONG
SIGNED LONGWORD SCALE n†	DECIMAL or any data type
SIGNED LONGWORD SCALE –n†	DECIMAL or any data type
SIGNED QUADWORD†	G_FLOATING
SIGNED QUADWORD SCALE n†	DECIMAL or any data type
SIGNED QUADWORD SCALE –n†	DECIMAL or any data type
F_FLOATING	SINGLE
G_FLOATING	G_FLOATING
DATE†	STRING 8
TEXT n†	STRING n
VARYING STRING n†	STRING n
SEGMENTED STRING †	Unsupported†

[1]The “n” stands for an integer value.

[2]If a data type is flagged by a dagger, † you cannot copy that definition into your program from the data dictionary.

The SEGMENTED STRING data type is not supported in BASIC. However, you can still use segmented strings in RDBPRE BASIC programs. See Chapter 13 for details.

Table 8–5 RDML Data Type Conversions for C

If the Rdb/VMS Data Type Is:	Declare Your C Variable as:
SIGNED BYTE	char
SIGNED BYTE SCALE n†[1]	int n (when 1< = n => 4) char[8](when n>4)
SIGNED BYTE SCALE –n†[2]	float
SIGNED WORD	short
SIGNED WORD SCALE n†[1]	int n (when 1< = n => 4) char[8](when n>4)
SIGNED WORD SCALE –n†[2]	float
SIGNED LONGWORD	int
SIGNED LONGWORD SCALE n†	char[8]
SIGNED LONGWORD SCALE –n†	double
SIGNED QUADWORD†	char[8]
SIGNED QUADWORD SCALE n†	char[8]
SIGNED QUADWORD SCALE –n†	double
F_FLOATING	float
G_FLOATING	double
DATE	char[8]
TEXT n	char [n+1]
VARYING STRING n†	Unsupported
SEGMENTED STRING †	char[8]

[1]The “n” stands for an integer value.

[2]If you want to copy this definition into your program from the data dictionary, you should check the conversion performed by the data dictionary and make sure that it is appropriate for your application.

The SEGMENTED STRING data type is not supported in C. However, you can still use segmented strings in RDML/C programs. See Chapter 17 for details.

Table 8–6 Rdb/VMS Data Type Conversions for COBOL

If the Rdb/VMS Data Type Is:	Declare Your COBOL Variable as:
SIGNED BYTE	Unsupported
SIGNED WORD	PIC S9(1-4) COMP
SIGNED WORD SCALE n[1]	PIC S9(n)P(n) COMP
SIGNED WORD SCALE –n	PIC S9(1-4)V9(n) COMP
SIGNED LONGWORD	PIC S9(5-9) COMP
SIGNED LONGWORD SCALE n	PIC S9(5-9)P9(n) COMP
SIGNED LONGWORD SCALE –n	PIC S9(5-9)V9(n) COMP
SIGNED QUADWORD	PIC S9(10-18) COMP
SIGNED QUADWORD SCALE n	PIC S9(10-18)P(n) COMP
SIGNED QUADWORD SCALE –n	PIC S9(10-18)V9(n) COMP
F_FLOATING	COMP-1
G_FLOATING†	PIC 9(10-18) COMP
DATE	PIC S9(11)V9(7) COMP
TEXT n	PIC X(n)
VARYING STRING n†	PIC X(n)
SEGMENTED STRING †	Unsupported†

[1]The “n” stands for an integer value.

Note that you should choose a value within the given range, where a range is indicated in Table 8–6. For example, a SIGNED WORD can be declared in COBOL as PIC S9(1) COMP, or PIC S9(2) COMP, and so on. Rdb/VMS will accept any value within the range given. You should decide which value to use on the basis of your COBOL needs.

The SEGMENTED STRING data type is not supported in COBOL. However, you can still use segmented strings in RDBPRE COBOL programs. See Chapter 14 for details.

Table 8–7 Rdb/VMS Data Type Conversions for FORTRAN

If the Rdb/VMS Data Type Is:	Declare Your FORTRAN Variable as:
SIGNED BYTE	BYTE or LOGICAL*1
SIGNED BYTE SCALE n†[1]	REAL*4
SIGNED BYTE SCALE –n†[2]	REAL*4
SIGNED WORD	INTEGER*2
SIGNED WORD SCALE n†[1]	REAL*4
SIGNED WORD SCALE –n†[2]	REAL*4
SIGNED LONGWORD	INTEGER or INTEGER*4
SIGNED LONGWORD SCALE n†	REAL*8
SIGNED LONGWORD SCALE –n†	REAL*8
SIGNED QUADWORD†	G_FLOATING
SIGNED QUADWORD SCALE n†	REAL*8
SIGNED QUADWORD SCALE –n†	REAL*8
F_FLOATING	REAL or REAL*4
G_FLOATING	REAL*8
DATE†	CHARACTER*8
TEXT n	CHARACTER*n
VARYING STRING n†	CHARACTER*n
SEGMENTED STRING †	Unsupported†

[1]The “n” stands for an integer value.

[2]If you want to copy this definition into your program from the data dictionary, you should check the conversion performed by the data dictionary and make sure that it is appropriate for your application.

The SEGMENTED STRING data type is not supported in FORTRAN. However, you can still use segmented strings in RDBPRE FORTRAN programs. See Chapter 15 for details.

Table 8–8 RDML Data Type Conversions for Pascal

If the Rdb/VMS Data Type Is:	Declare Your Pascal Variable as:
SIGNED BYTE	[BYTE] –128 . . . 127

SIGNED BYTE SCALE n†[1]	n=1,2:[WORD]–32768 . . . 32767
	n=3,4,5,6,7:INTEGER
	n>8:RECORD
	L0:UNSIGNED;
	L1:INTEGER;
	END

SIGNED BYTE SCALE –n†[2]	REAL
SIGNED WORD	[WORD]–32768 . . . 32767

SIGNED WORD SCALE n†[1]	INTEGER(n=1,2,3,4)
	RECORD
	L0:UNSIGNED;
	L1:INTEGER;END(n>4)

SIGNED WORD SCALE –n†[2]	REAL
SIGNED LONGWORD	INTEGER

SIGNED LONGWORD SCALE n†	RECORD
	L0:UNSIGNED;
	L1: INTEGER;
	END

SIGNED LONGWORD SCALE –n†	DOUBLE

SIGNED QUADWORD†	RECORD
	L0:UNSIGNED;
	L1: INTEGER;
	END

SIGNED QUADWORD SCALE n†	RECORD
	L0:UNSIGNED;
	L1: INTEGER;
	END

[1]The “n” stands for an integer value.

[2]If you want to copy this definition into your program from the data dictionary, you should check the conversion performed by the data dictionary and make sure that it is appropriate for your application.

(continued on next page)

Table 8–8 (Cont.) RDML Data Type Conversions for Pascal

If the Rdb/VMS Data Type Is:	Declare Your Pascal Variable as:
SIGNED QUADWORD SCALE –n†	DOUBLE
F_FLOATING	REAL
G_FLOATING	DOUBLE
DATE	[BYTE(8)]RECORD END
TEXT n	CHAR(n=1) PACKED ARRAY [1 . . . n] OF CHAR (n>1)
VARYING STRING n	VARYING[n] OF CHAR

SEGMENTED STRING †	RECORD
	L0:UNSIGNED;
	L1: INTEGER;
	END

The SEGMENTED STRING data type is not supported in Pascal. However, you can still use segmented strings in RDML/Pascal programs. See Chapter 18 for details.

9 Program Structure and Design

The basic structure for an Rdb/VMS program is the same regardless of whether you use the RDML or the RDBPRE preprocessor, or the Callable RDO interface. Each program must attach to a database and start a transaction before performing any data manipulation or data definition tasks. Each transaction must be either committed or rolled back. Each program must detach from the database (with a FINISH statement) before it ends. Furthermore, as with any programming language, you need to pay attention to design aspects to ensure that your program executes quickly and accurately.

Performing data manipulation and data definition tasks may involve some or all of the following functions:

- Using host language variables to pass values between your program and Rdb/VMS
- Accessing one or more databases
- Starting a transaction
- Designing a transaction that will fit your needs and minimize contention for the database (locking considerations)
- Forming record streams so that you can:
  - Retrieve records
  - Update records
  - Store records
  - Erase records
- Using structured programming techniques
- Using database handles, transaction handles, and request handles
- Handling Rdb/VMS run-time errors

- Rolling back transactions
- Committing transactions
- Detaching from the database

See the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for information on locking. For information on handling Rdb/VMS run-time errors, see Chapter 10. The other topics are discussed in this chapter.

Note *If you create an Rdb/VMS application that runs as a detached process, you must define permanent logical names for the process (in particular, SYS$LOGIN and SYS$SCRATCH), at the system or group level by using the DCL DEFINE command or defining them within your program. Permanent logical names are not defined by default for detached processes. SYS$LOGIN is used typically for recovery-unit journal files and bugchecks; SYS$SCRATCH is used for the temporary files created by the VMS Sort utility.*

9.1 Embedding DML Statements in the Program Environment

The rest of this chapter discusses how to use data manipulation language (DML) statements in the RDBPRE and RDML programming environment. The purpose of this section is to explain the design implications of using the RDBPRE languages (BASIC, COBOL, or FORTRAN) and the RDML languages (C and Pascal). All examples are pseudocode; however, the language-specific chapters demonstrate how to use the concepts explained in this chapter in your host language program.

Note that error handling is not covered in this chapter. Keep in mind that all the executable DML statements and clauses permit the use of the ON ERROR clause to trap error conditions that occur during the execution of an RDML or RDBPRE statement or clause.

Error handling is discussed in general in Chapter 10. Language-specific error handling issues are discussed in the language-specific chapters.

9.2 Declaring Host Language Variables

A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. You can use host language variables as value expressions in data manipulation statements, as well as for any other program function.

When you declare host language variables, simply follow the naming rules for your language. Ensure that host language variable data types and sizes are compatible with the corresponding database field data types and sizes. You can declare variables by:

- Using the methods that you usually use to declare host language variables in your host language
- Using the DECLARE_VARIABLE and BASED ON clauses in RDML programs

  These clauses declare variables for you with data types and sizes that are compatible with the corresponding database field data types and sizes.
- Copying database definitions from the CDD/Plus data dictionary

  You can copy relation definitions, which include all the fields within the relation. However, you must be careful to copy only those relation and field definitions with data types that are supported by your host language.

See Chapter 8 for information on how to select host language data types that are compatible with Rdb/VMS data types.

See Chapter 16 for information on the DECLARE_VARIABLE and BASED ON clauses in RDML programs.

For more information on using the data dictionary see Chapter 12 and Chapter 16.

You can use host language variables in:

- Any data manipulation statement that can include an RSE
- Any of the update statements
- The GET statement

Note *The RDBPRE preprocessor and Callable RDO interpret a hyphen between two variables or strings (with no intervening spaces) as an underscore. For example, A-B is interpreted as A_B. When you want a hyphen to be interpreted as a hyphen, leave a blank space on each side of it. For example, A - B.*

For more information on using host language variables, see the language-specific chapter for the host language you are using.

9.2.1 Declaring Databases

The first Rdb/VMS statement in your program must be the DATABASE statement. The DATABASE statement is a nonexecutable statement that declares the database to your program.

Note that the DATABASE statement does not cause an attach to the database in RDBPRE or RDML programs. However, it does cause an attach to the database in Callable RDO programs. (For more information on Callable RDO, see Chapter 19.)

Additionally, the DATABASE statement lets you specify a variable to identify the database; this is called a **database handle**. If you do not explicitly supply a database handle, Rdb/VMS uses the default database handle. The chosen handle can be seen in the output from the RDML or RDBPRE preprocessor. Rdb/VMS uses the database handle to identify the particular database that is referred to by a database request. Your program must not alter the database handle.

Both RDML and RDBPRE programs attach to the database at the READY statement. If a READY statement is not specified, both RDML and RDBPRE will start a transaction at the first executable statement after the DATABASE statement. However, Digital recommends that you always use the READY statement. The READY statement makes your intentions obvious to others who might use your program, and in RDML, can reduce program overhead *when used in conjunction with the /NODEFAULT_TRANSACTIONS qualifier* on the RDML command line. (For more information on the /NODEFAULT_ TRANSACTIONS qualifier, see Chapter 11.)

If your program invokes multiple databases, use database handles in each DATABASE statement. Using database handles lets you attach to the same database more than once, invoke multiple databases in the same program, and refer to each database attach.

When your program accesses a single database you do not have to include a database handle in the DATABASE statement. Use either the file name or the path name to specify the location of the database. Enclose the file name or path name in single or double quotation marks.

See Section 9.3.3 for more information on database handles and scope. In Callable RDO programs, you must pass the database handle as a parameter to the RDB$INTERPRET function in the FINISH statement.

When you no longer need the resources for a particular database, use the FINISH statement. The FINISH statement causes Rdb/VMS to detach from the database, and all variables connected to that database become undefined. However, when Rdb/VMS attaches to a database, it loads metadata for that database into memory. Be careful about detaching and reattaching to a database too frequently; reattaching incurs significant overhead because the metadata must be loaded again.

To detach from a particular database when multiple databases have been declared, use the same database handle in the FINISH statement that you used in the DATABASE statement for that database, as shown in the following example:

```
DATABASE PERSONNEL = FILENAME 'MF_PERSONNEL'
DATABASE PAYROLL = PATHNAME 'PAYROLL'

   READY PAYROLL
       .
       .
       .
   READY PERSONNEL
       .
       .
       .
   FINISH PERSONNEL
       .
       .
       .
   FINISH PAYROLL
```

9.2.2 Forming Record Streams

Rdb/VMS data manipulation statements use context variables to form a record stream of selected records in one or more relations. The RSE may be as simple as FOR E IN EMPLOYEES, or as complex as:

```
FOR E IN EMPLOYEES CROSS JH IN JOB_HISTORY
  CROSS SE IN EMPLOYEES CROSS SJH IN JOB_HISTORY
    WITH JH.JOB_END MISSING
      AND JH.EMPLOYEE_ID = E.EMPLOYEE_ID
      AND E.EMPLOYEE_ID = '00205'
      AND SJH.JOB_END MISSING
      AND SJH.EMPLOYEE_ID = SE.EMPLOYEE_ID
      AND SE.EMPLOYEE_ID = JH.SUPERVISOR_ID
```

The better RSE is generally the more efficient one. Rdb/VMS lets you define views, which save preprocessing time and assist in the optimization of the query. When you define a view, the database stores the view definition in BLR. This last query could be formed more efficiently by using the view CURRENT_JOB. For example:

```
FOR CJ IN CURRENT_JOB
  CROSS CJ2 IN CURRENT_JOB
    WITH CJ.EMPLOYEE_ID = '00205'
    AND CJ2.EMPLOYEE_ID = CJ.SUPERVISOR_ID
```

The maximum number of subqueries or relation references in an Rdb/VMS statement is 32. A subquery is a nested FOR statement. A relation reference can be any of the entities in the following list:

- A relation in an RSE
- A global aggregate
- A relation in a view

9.2.3 Retrieving Records

Rdb/VMS provides you with two ways to form record streams for retrieving records:

- Using the FOR statement

  The FOR statement forms a record stream and provides automatic iteration for any Rdb/VMS and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

- Using one of the START_STREAM statements

  The START_STREAM statements also form record streams, but do not provide automatic iteration of any Rdb/VMS or host language statements. The START_STREAM statements give you total control of program iteration. Your host language statements must provide all the control logic for processing the stream.

  Rdb/VMS provides two kinds of streams:

  – Declared streams

  – Undeclared streams

The following sections describe how to form record streams using the FOR and the two START_STREAM statements.

9.2.3.1 Using the FOR Statement to Retrieve Records

The FOR statement forms a record stream and provides automatic iteration for any DML and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

The scope of the context variable begins with the FOR statement and ends with the END_FOR statement. A context variable is meaningless outside its scope; thus, you need to consider the scope of context variables when designing structured programs.

If Rdb/VMS does not find any record that satisfies the conditions of the RSE in the FOR statement, the FOR loop is not executed. Rdb/VMS does not treat this as an exception condition. Therefore, if you want to detect this condition, set a flag within the FOR block. Your program can evaluate the flag immediately after the END_FOR statement to determine if the RSE has been satisfied and the loop executed.

In the following example, the uppercase statements are DML statements. Lowercase statements must be converted to your host language. If you code this example into your host language, you would want your program to:

- Set the host language variable, record_found, to false.
- Begin a FOR . . . END_FOR statement.

- Set the host language variable, record_found, to true and modify the record if a record is found that matches the RSE.
- Display a message to the user if a record is not found that meets the requirements of the RSE. If no record is found, the host language variable remains false and no modification of an employee record takes place.

```
record_found = false
      FOR E IN EMPLOYEES CROSS JH IN JOB_HISTORY
           WITH JH.JOB_END MISSING
           AND JH.EMPLOYEE_ID = E.EMPLOYEE_ID
              MODIFY JH USING
                JH.DEPARTMENT_CODE = new_dept_code
              END_MODIFY
              record_found = true
      END_FOR
if record_found = false then
 print "No record found on modify"
```

You can include host language statements within the FOR . . . END_FOR block to process the records within the stream. However, there are some important exceptions to the type of statement you can include:

- Do not transfer control out of the FOR . . . END_FOR block unless you do not want to return to the FOR loop. It is impossible to enter the loop again while it is executing.
- You may call a procedure within a FOR loop because such a procedure executes within the FOR loop context. However, do not use a context variable defined in the FOR block in any statement outside the context of the FOR block.

9.2.3.2 Using Streams to Retrieve Records Rdb/VMS provides two kinds of streams that are opened with a START_STREAM statement; a declared stream and an undeclared stream. A **declared stream** is one that you explicitly declare in your program with the DECLARE_STREAM statement. The DECLARE_STREAM statement includes an RSE (and transaction and request handles, or both, if you choose to use them). Therefore the START_STREAM statement for a declared stream does not include an RSE, a transaction handle, or a request handle. However, it must be preceded by the DECLARE_STREAM statement. For example:

```
DECLARE_STREAM cands USING
    CA IN CANDIDATES SORTED BY CA.LAST_NAME;
            .
            .
            .
START_STREAM cands;
```

An **undeclared stream** does not use the DECLARE_STREAM statement. Instead, you specify the RSE (and transaction and request handles, or both, if you choose to use them) on the START_STREAM statement. For example:

```
START_STREAM cand USING CA IN CANDIDATES SORTED BY CA.LAST_NAME;
                .
                .
                .
END_STREAM cand;
```

Which kind of stream you use in your program will have an impact on how your program must be structured and how the streams can be manipulated. When you use a declared stream, the declaration of your stream made with the DECLARE_STREAM statement is valid for the duration of your program. This means that:

- The location of the START_STREAM, FETCH, GET, and END_STREAM statements in your program is flexible.

  You can place these statements in any order within your source program as long as they are preceded by the DECLARE_STREAM statement and execute in a logical order. That order is:

  1. START_STREAM statement
  2. FETCH statement
  3. GET statement or a host language assignment to a variable in RDML
  4. END_STREAM statement

- The context variables you use in the DECLARE_STREAM statement are meaningful only at preprocessing time and endure until the end of the module being processed. You cannot use a context variable referred to in a DECLARE_STREAM statement in a FOR statement, for example.
- You can issue several START_STREAM statements, and as long as you use the declared stream name, they will all manipulate the same stream.

An undeclared stream does not offer the flexibility of a declared stream. When you use an undeclared stream, the context variables specified in the START_STREAM statement are only valid for code that physically appears between the START_STREAM and END_STREAM statements in your source program.

This means that stream manipulation statements must appear in the source program in exactly this order:

1. START_STREAM statement
2. FETCH statement

3 GET statement or a host language assignment to a variable in RDML

4 END_STREAM statement

Because declared streams offer all the functionality of undeclared streams, and allow you more flexibility in programming, Digital recommends that you use declared streams rather than undeclared streams.

9.2.3.3 Using Undeclared Streams to Retrieve Records You can use an undeclared stream to retrieve all or some of the records from a record stream; you have total control of program iteration. With an undeclared START_STREAM statement you can conditionally terminate processing the record stream at any time, a feature not available with the FOR statement.

When you use an undeclared stream, you must name the stream and specify an RSE with at least one context variable. The RSE operates in exactly the same way as it does in the FOR loop; it determines which records and which fields from those records are included in the record stream. For detailed information about forming RSEs, refer to Chapter 3, and to the *VAX Rdb/VMS RDO and RMU Reference Manual* and the *RDML Reference Manual*.

When you issue an undeclared START_STREAM statement that contains host language variables in the RSE, Rdb/VMS examines the host language variables at the time it executes the statement. Any changes you make to the host language variables after execution of the START_STREAM statement have no effect on the records included in the stream.

After starting a record stream with the START_STREAM statement, use the FETCH statement to step through the stream and a GET statement (or in RDML, a host language assignment statement) to transfer the database value to a host language variable. Use host language statements to process the record retrieved by the GET statement.

Your program does not have to detect the end-of-stream condition explicitly. The FETCH statement includes an optional AT END clause to detect this condition. If the end-of-stream condition is detected, control passes to the host language statements within the AT END . . . END_FETCH block.

The ON ERROR clause in the FETCH statement can handle errors or exception conditions other than end-of-stream. When an error occurs, program control is transferred to the statements you include in the ON ERROR clause.

You can process a record stream only from the beginning. To return to a record you have already processed, you must first end the stream and then start it again. To end a stream, issue an END_STREAM statement that includes the same stream name used to start the stream. Do not issue an END_STREAM statement after a COMMIT or ROLLBACK statement. The COMMIT and ROLLBACK statements automatically end all streams opened during that transaction.

The START_STREAM statement always includes at least one context variable in its RSE. This context variable is valid starting with the START_STREAM statement and ending with the END_STREAM statement. The context variable associated with an undeclared stream is meaningless when you refer to it outside of the block of code enclosed by the START_STREAM and END_STREAM statements in your source program.

If you do not include an END_STREAM statement for a particular record stream, the context variable is valid to the end of the transaction. You should consider this when you design structured programs.

Host language statements within the START_STREAM . . . END_STREAM block can process the records within the stream.

You can call procedures from within a START_STREAM . . . END_STREAM block; these procedures can also form streams. However, if the calling procedure has an open stream, and the procedure uses any of the same context variables in its RSE, you will receive an error. You will receive the error at preprocessing time if the calling procedure and the called procedure are in the same module. If the calling procedure and the called procedure are in separately preprocessed modules, you will receive an error at run time.

You may use the START_STREAM statement within a FOR loop. However, *in RDBPRE programs only*, you will receive the error RDB$_REQ_SYNC if you attempt to fetch a field defined in the FOR statement RSE more than once inside the stream. The following example generates the RDB$_REQ_SYNC error because it attempts to retrieve the variable P.CITY (used in the FOR loop RSE) more than once inside the stream named S:

```
        MOVE 'N' TO FOUND_END
&RDB& FOR P IN PORT WITH P.CITY MATCHING '*D*'
&RDB&   START_STREAM S USING E IN EXPORTER WITH
&RDB&      E.PORT_NUM = P.PORT_NUM

        PERFORM UNTIL FOUND_END IS EQUAL 'Y'
&RDB&     FETCH S
&RDB&       AT END
              DISPLAY " End of stream found "
              MOVE 'Y' TO FOUND_END
&RDB&     END_FETCH
*
* Need conditional statement to branch around
* this code at stream end.
*
```

```
        IF FOUND_END IS EQUAL TO ’N’ THEN
*
* Keep retrieving the field value of P.CITY until
* the end of stream is reached.  The second attempt
* to retrieve this field will result in an error.
*
&RDB& GET
&RDB&     CITY = P.CITY;
&RDB&     NAM  = E.EXP_NAME; END_GET

        DISPLAY "City  ",CITY,"  Exporter  ",NAM
        END-IF

        END-PERFORM

        MOVE ’N’ TO FOUND_END
&RDB& END_STREAM S
&RDB& END_FOR
```

Instead, retrieve the variable used in the FOR loop RSE outside the START_STREAM . . . END_STREAM block. For example:

```
        MOVE ’N’ TO FOUND_END
&RDB& FOR P IN PORT WITH P.CITY MATCHING ’*D*’
&RDB&    GET
&RDB&      CITY = P.CITY;
&RDB& END_GET

&RDB&   START_STREAM S USING E IN EXPORTER WITH
&RDB&      E.PORT_NUM = P.PORT_NUM

        PERFORM UNTIL FOUND_END IS EQUAL ’Y’
&RDB&     FETCH S
&RDB&        AT END
                DISPLAY " End of stream found "
                MOVE ’Y’ TO FOUND_END
&RDB&     END_FETCH

*
* Need conditional statement to branch around
* this code at stream end.
*
        IF FOUND_END IS EQUAL TO ’N’ THEN
&RDB& GET
&RDB&      NAM  = E.EXP_NAME;
&RDB& END_GET

        DISPLAY "City  ",CITY,"  Exporter  ",NAM
        END-IF

        END-PERFORM

        MOVE ’N’ TO FOUND_END
&RDB& END_STREAM S
&RDB& END_FOR
```

The following example shows the use of an undeclared stream in pseudocode. Uppercase statements are DML statements. Lowercase text describes the logic you should code in your host language. The example:

- Creates a stream of all EMPLOYEES records sorted by the LAST_NAME field
- Creates a stream of all EMPLOYEES records sorted by the FIRST_NAME field
- Uses the FETCH statement to step through the LAST_NAME stream, record by record
- Uses the FETCH statement to step through the FIRST_NAME stream, record by record
- Uses a GET statement to retrieve a record from the LAST_NAME stream, then a record from the FIRST_NAME stream so that a host language display statement can be used to list the stream sorted by LAST_NAME in the left columns and the stream sorted by FIRST_NAME in the righthand columns.

```
START_STREAM BY_LAST_NAME USING
   E1 IN EMPLOYEES SORTED BY E1.LAST_NAME, E1.FIRST_NAME;
START_STREAM BY_FIRST_NAME USING
   E2 IN EMPLOYEES SORTED BY E2.FIRST_NAME, E2.LAST_NAME;

   set flag for end-of-stream to false

   FETCH BY_LAST_NAME
      AT END
         set end-of-stream flag to true
   END_FETCH;

   if not end-of-stream then
      FETCH BY_FIRST_NAME

   while end-of-stream = false
      begin loop
         GET
            last_name =  E1.LAST_NAME;
            first_name = E2.FIRST_NAME;
            print E1.LAST_NAME and E1.FIRST_NAME
            then skip 20 spaces
         END_GET

         FETCH BY_LAST_NAME
            AT END
               set end-of-stream flag to true
            END_FETCH;

         if not end-of-stream then
            FETCH BY_FIRST_NAME;

      end loop
```

```
END_STREAM BY_LAST_NAME;
END_STREAM BY_FIRST_NAME;
```

9.2.3.4 Using Declared Streams to Retrieve Records

As with an undeclared stream, a declared stream allows you total control of program iteration. That is, you can conditionally terminate processing of the record stream at any time, a feature not available with the FOR statement.

However, when you use a declared stream, you must also use a DECLARE_STREAM statement. In the DECLARE_STREAM statement, you must name the stream and specify an RSE with at least one context variable. Then, when you start the stream with the START_STREAM statement, you must use the same name to refer to the stream as you specified in the DECLARE_STREAM statement.

In a declared stream (as with the undeclared stream) the RSE operates in exactly the same way as it does in the FOR loop; it determines which records your program processes. For detailed information about forming RSEs, refer to Chapter 3, and to the *VAX Rdb/VMS RDO and RMU Reference Manual* and the *RDML Reference Manual*.

When you use a DECLARE_STREAM statement that contains host language variables in the RSE, Rdb/VMS examines the host language variables at the time it executes the declared START_STREAM statement. Any changes you make to the host language variables after the execution of the declared START_STREAM statement have no effect on the records included in the stream formed by the DECLARE_STREAM statement.

After opening a record stream with the START_STREAM statement, use the FETCH statement to step through the stream and a GET statement, (or in RDML, a host language assignment statement) to transfer the database value to a host language variable. Use host language statements to process the record retrieved by the GET statement.

Your program does not have to detect the end-of-stream condition explicitly. The FETCH statement includes an optional AT END clause to detect this condition. If the end-of-stream condition is detected, control passes to the host language statements within the AT END . . . END_FETCH block. If you use the AT END clause, you must use the END_FETCH clause to terminate the FETCH statement.

The ON ERROR clause in the FETCH statement can handle errors or exception conditions other than end-of-stream. When an error occurs, program control is transferred to the statements you include in the ON ERROR clause.

To end a declared stream, issue the END_STREAM statement, which must include the same stream name used to start the stream. You do not need to issue an END_STREAM statement after a COMMIT or ROLLBACK statement. The COMMIT and ROLLBACK statements automatically close all streams opened during that transaction.

The DECLARE_STREAM statement always includes at least one context variable in its RSE. This context variable is valid starting with the DECLARE_ STREAM statement to the end of the module. Note that this differs from an undeclared stream, in which the validity of a context variable begins with the START_STREAM statement and ends with the END_STREAM statement.

Note that a declared stream name cannot be passed between separately preprocessed modules. Although the DECLARE_STREAM, START_STREAM, FETCH, GET, and END_STREAM statements may appear within different procedures of a module, the modules must appear within the same source file.

The following example shows the use of a declared stream in pseudocode. Uppercase statements are DML statements. Lowercase text describes the logic you should code in your host language. The example:

- Declares and specifies an RSE for the *cands* stream using the DECLARE_ STREAM statement
- Declares and specifies an RSE for the *emps* stream using the DECLARE_ STREAM statement
- Uses declared START_STREAM statements to open the cands and emps streams
- Uses a FETCH statement to place a pointer at the first record in the cands record stream
- If a record exists in the cands stream, uses the GET statement to place the values from the LAST_NAME, FIRST_NAME, and CANDIDATE_STATUS fields into host language variables
- Uses a FETCH statement to place a pointer at the first record in the emps record stream
- If a record exists in the emps stream, uses the GET statement to place the values from the LAST_NAME, FIRST_NAME, and EMPLOYEE_ID fields into host language variables
- Displays all the values that the GET statement placed in the host language variables
- Continues to fetch, get, and display records until there are no more records in the cands record stream

```
DECLARE_STREAM cands USING CA IN CANDIDATES
   SORTED BY CA.LAST_NAME
DECLARE_STREAM emps USING EM IN EMPLOYEES
   SORTED BY EM.FIRST_NAME

START_TRANSACTION READ_ONLY

   START_STREAM cands
   START_STREAM emps
```

```
set flag for end of emps stream (emps_end) to false
set flag for end of cands stream (cands_end) to false

FETCH cands
   AT END
      set flag for end of the cands stream to true
END_FETCH

if it is not the end of the cands stream then

   GET
      cand_last_name = CA.LAST_NAME;
      cand_first_name = CA.FIRST_NAME;
      cand_status = CA.CANDIDATE_STATUS;
   END_GET

end of if statement block

FETCH emps
   AT END
      set flag for end of the emps stream to true
END_FETCH

if it is not the end of the emps stream then

   GET
      last_name = EM.LAST_NAME;
      first_name = EM.FIRST_NAME;
      employee_id = EM.EMPLOYEE_ID;
   END_GET

end of if statement block

execute the following loop as long as it is not
the end of the cands stream

begin loop
   display  last_name,first_name,
            cand_last_name,cand_first_name
      FETCH cands
         AT END
            set flag for end of
            the cands stream to true
      END_FETCH

      if the flag for the end of the cands stream
        is set to false then do the following:
         begin loop
         GET
            cand_last_name = CA.LAST_NAME;
            cand_first_name = CA.FIRST_NAME;
            cand_status = CA.CANDIDATE_STATUS;
         END_GET

      end of if statement block

      if the flag for the end of the emps stream
      is set to false, then do the following:
            FETCH emps
               AT END
                  set flag for end of emps
                  stream to true
            END_FETCH

            if the flag for the end of the emps stream
            is set to false then do the following:

               GET
                  last_name = em.last_name;
                  first_name = em.first_name;
                  employee_id = em.employee_id;
               END_GET
            end of inner if statement block
         end of outer if statement block
   end loop

   END_STREAM emps
   END_STREAM cands

COMMIT
```

9.2.4 Retrieving Segmented Strings

The Rdb/VMS segmented string data type allows you to store blocks of unstructured data such as text, graphics, or voice. You store segmented string records in a field of a relation. Each record can hold any number of segmented strings, up to the physical limits of the storage unit. Each segment can be up to 65,522 bytes long, except for the first segment of the string, which has a maximum length of 65,508 bytes. See Chapter 8 for more information on the segmented string data type.

The Rdb/VMS segmented string data type requires a special use of RSEs. The first RSE forms an outer stream of records. It determines the field and the relation that will contain the segmented string records. A second RSE forms the inner stream of segments. It identifies the segmented string field that contains the individual segments.

The RDBPRE preprocessor lets you use either the START_SEGMENTED_STRING statement or a FOR statement with segmented strings to form a stream of segmented string records.

RDML allows only the FOR statement with segmented strings; RDML does not support the START_SEGMENTED_STRING statement.

9.2.4.1 Using the FOR Statement to Retrieve Segmented Strings

You must use either two nested FOR statements or an outer START_STREAM statement with an inner FOR statement to create two streams when retrieving segmented string records. The inner RSE identifies the segments contained in the field specified by the outer RSE. Use a different context variable in the inner and outer FOR or START_STREAM statements.

The inner RSE is not an RSE in the sense that it can select records. The segmented string behaves like a sequential record file. You must begin at the first segment and retrieve segments in the order that they are stored. For this reason, the inner RSE does not include selection clauses. Note that the inner FOR statement uses a segmented string variable in place of the context variable, and that the field name is qualified by the context variable specified in the outer FOR statement.

There are two special variables recognized by the preprocessors: RDB$VALUE and RDB$LENGTH. The RDB$VALUE variable contains the segmented string segment just retrieved. The RDB$LENGTH variable is a signed word integer that contains the length of this segment. Within the inner loop, the GET statement automatically fetches the contents of the segment, RDB$VALUE. Note that you can have only one GET . . . END_GET block within the inner loop.

The following pseudocode demonstrates the logic you should use to retrieve a segmented string. The DML statements are in uppercase. Statements that appear in lowercase must be translated into your host language. This example:

- Starts a read-only transaction
- Uses a FOR statement to start an outer stream of records that will include all records for the employee with the employee ID specified by a host language variable, employee_id
- Uses a second FOR statement to start a stream of segments that form a segmented string
- Uses a GET statement to retrieve the value of the segmented string, segment by segment
- Prints each segment of the segmented string with a host language print statement
- Ends the inner FOR statement

- Ends the outer FOR statement
- Commits the transaction

```
START_TRANSACTION READ_ONLY
   FOR R IN RESUMES WITH
   R.EMPLOYEE_ID = employee_id
   set flag employee_found to true
      FOR RR IN R.RESUME
         GET
            resume_segment = RR.RDB$VALUE;
            segment_length = RR.RDB$LENGTH;
         END_GET

         print resume_segment
      END_FOR   !Ends inner FOR statement
   END_FOR           !Ends outer FOR statement
COMMIT

if flag employee_found is set to false then
display "Employee has no resume on file"
```

9.2.4.2 Using the START_SEGMENTED_STRING Statement to Retrieve Segmented Strings

When you want to maintain program control of a stream of segments that comprise the segmented string field, use the START_SEGMENTED_STRING statement instead of using the FOR statement.

Note *This statement is only available in RDBPRE. Do not attempt to use it in RDML programs.*

You must start two streams when processing segmented strings with the START_SEGMENTED_STRING statement. One stream (the outer stream) retrieves the records that you specify; the other stream (the inner stream) is comprised of the segments that form the segmented string.

Form an outer stream of records with the FOR or START_STREAM statement, then use the START_SEGMENTED_STRING statement to form an inner stream of segments. The inner stream identifies the segments that are contained in the field specified by the FOR or START_STREAM statement. Use different context variables for the outer record stream and the inner record stream.

The inner stream is not a stream in the sense that you can control its record selection. The segmented string behaves like a sequential record file. You must begin at the first segment and retrieve segments in the order that they are stored. For this reason, the inner stream does not include selection clauses. Note that the START_SEGMENTED_STRING statement uses a segmented string variable in place of the context variable, and that the field name is qualified by the context variable specified in the outer record stream.

When you use the START_STREAM statement, use the FETCH statement to advance the pointer in the outer record stream. (The outer record stream advances automatically with the FOR statement.) Use the GET statement in the inner stream to retrieve each segment, RDB$VALUE, in the segmented string. Within the START_SEGMENTED_STRING statement, the GET statement automatically retrieves the segmented string, segment by segment.

The following example uses the START_STREAM statement to form a stream of resumes records that have an EMPLOYEE_ID field value of 12345. Then, an inner START_SEGMENTED_STRING statement is used to form the stream of segments that form the segmented string. A host language loop is used to control the processing of the segmented strings. In this example, the stream is processed until the last segment has been retrieved from the database. Each segment of the segmented string is printed until the end-of-segmented-string condition is met (SEGSTR_EOF). When this condition is met, the stream created by the START_SEGMENTED_STRING statement is closed, and if there are no more records with the EMPLOYEE_ID of 12345, the stream formed by the START_STREAM statement is closed also. (Remember that when you use the START_SEGMENTED_STRING statement, any host language conditional statement can be used to determine when the program should stop processing the segmented string stream; you do not have to use the RDB$_SEGSTR_EOF condition as the terminating condition.)

```
DATABASE pers = FILENAME 'MF_PERSONNEL'

set flag for end of segmented string stream to false

   START_TRANSACTION READ_ONLY
      START_STREAM RESSTR USING
         R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
            FETCH RESSTR
            END_FETCH

         START_SEGMENTED_STRING RINFO USING STRN IN R.RESUME
            loop until end of segmented strings
               GET
                  ON ERROR
                     call error handler
                  END_ERROR
                  resume_segment = STRN.RDB$VALUE;
                  segment_length = STRN.RDB$LENGTH;
               END_GET

               trap status of GET statement.

               if status is success print resume_segment
```

```
              if status equals RDB$_SEGSTR_EOF then
               set end_of_seg flag to true
           end loop
        END_SEGMENTED_STRING RINFO
     END_STREAM RESSTR
  COMMIT
```

9.2.5 Retrieving Field and Statistical Values

Use the GET statement to retrieve one, several, or all the fields in a database record. You can also use a GET statement that contains a statistical function to retrieve the following statistical values from the database:

- Total value of the selected fields (TOTAL)
- Minimum value of the selected fields (MIN)
- Maximum value of the selected fields (MAX)
- Average value of the selected fields (AVERAGE)
- Count of the selected fields (COUNT)

The next two sections discuss retrieving field and record values, and retrieving statistical values.

9.2.5.1 Using the GET Statement to Retrieve Field and Record Values

When you form a record stream using the FOR statement, you include the GET statement within the FOR . . . END_FOR block to place the database values in host language variables. In RDBPRE programs, the GET statement is the only way to place field values into host language variables. In RDML, you can either use the GET statement or a host language assignment statement to place field values in host language variables.

When you use an undeclared stream, you must use the GET statement to retrieve field values in RDBPRE programs (or in RDML, you can also use a host language assignment statement). However, the GET statement (or host language assignment statement) must appear in your source program after the FETCH statement and before the END_STREAM statements.

When you use a declared stream, the GET statement (or host language assignment statement) may appear anywhere within your source program, as long as it executes after the FETCH statement and before the END_STREAM statement.

Note that in RDML programs, use of the GET statement to retrieve the results of statistical and Boolean functions is recommended but not required. You can use a host language assignment statement in place of the GET statement. This is discussed in more detail in Chapter 17 and Chapter 18.

A special form of the GET statement is the GET * statement, which lets you retrieve all fields in a record. You can retrieve all the fields in the records of a relation with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the records of a relation, with record field names that match the database field names. You can create such a record structure by copying data definitions from the data dictionary. (See Chapter 12 and Chapter 16 for more information on copying record and field definitions from the data dictionary.) The following GET * statement retrieves all of the fields from the records of the JOB_HISTORY relation and places their values in the JOB_HISTORY record structure:

```
FOR FIRST 1 J IN JOB_HISTORY WITH
   J.JOB_CODE = JOB_CODE IN JOB_HISTORY
   AND J.JOB_END MISSING
      GET
         JOB_HISTORY = J.*
      END_GET
END_FOR
```

9.2.5.2 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly, without processing each record in the record stream. Statistical expressions are sometimes called aggregate expressions because they calculate a single value for a collection of records. RDBPRE or RDML may assign a data type to the result that is different from the data type of the field referred to in the expression. See Chapter 8 for information on the data type conversions performed by statistical expressions.

The following example uses the COUNT statistical function to find the total number of employees in the EMPLOYEES relation. To use this code in a host language program you need to convert the statements that appear in lowercase into your host language. Statements that appear in uppercase type are DML statements.

```
START_TRANSACTION READ_ONLY

   GET
      number_employees = COUNT OF E IN EMPLOYEES
   END_GET

   display number_employees
COMMIT
```

9.2.6 Updating Records Using the STORE, MODIFY, and ERASE Statements

The Rdb/VMS update statements can only be used in a read/write transaction. (You may, of course, include any valid Rdb/VMS statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

Note *You may not use a view to update records if that view refers to more than one relation.*

If your program prompts for data from a terminal, consider nesting a complete transaction within the data input loop. For example, you could use the following algorithm:

- Start a host language loop
- Request data input from user
- Start a transaction
- Perform data manipulation tasks
- Commit the transaction
- End the host language loop

If the transaction fails, the user needs to enter again only the last input data. However, do not place requests for input from the user within the scope of the transaction.

Keep in mind that the transaction should be short and, at the same time, the logic should ensure that all updates that logically go together are included in the same transaction. For example, if you want to delete an employee's records from the entire database, do not erase all of this employee's records from one relation during one transaction, and the rest of his or her records in the database during another transaction.

Examples of prompting for data from a terminal are in the language-specific chapters.

The following sections describe how to update records using the STORE, MODIFY, and ERASE statements.

9.2.6.1 Storing Records You can insert values into one or more fields in one record using a single STORE statement. To store more than one record in a relation, assign values in the STORE statement with host language variables and include the STORE statement within a program loop that assigns values again to host language variables before the STORE statement is issued.

The following example demonstrates the use of the STORE statement to add a new employee to the database. In this example, lowercase values are host language variables. This example starts a read/write transaction, the only type of transaction in which you can use a STORE operation. The STORE statement inserts a record with the values specified in the host language variables into the EMPLOYEES relation.

```
START_TRANSACTION READ_WRITE NOWAIT RESERVING
   EMPLOYEES FOR SHARED WRITE
      STORE E IN EMPLOYEES USING
         E.EMPLOYEE_ID = employee_id;
         E.LAST_NAME = last_name;
         E.FIRST_NAME = first_name;
         E.MIDDLE_INITIAL = middle_initial;
         E.ADDRESS_DATA_1 = address_data_1;
         E.ADDRESS_DATA_2 = address_data_2;
         E.CITY = city;
         E.STATE = state;
         E.POSTAL_CODE = postal_code;
         E.BIRTHDAY = birthday;
      END_STORE
COMMIT
```

A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in the records of a relation with the STORE * statement. To use the STORE * statement, you must first declare a record structure that contains all the fields in the relation, with record field names that match the database field names. You can copy record definitions from the data dictionary to create such a record structure. (See Chapter 12 and Chapter 16 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to store in the record fields and store the entire record using the STORE * statement. The following example shows the use of the STORE * statement to store the values in the host language record, job_log, in the JOB_HISTORY relation of the PERS database:

```
STORE J IN PERS.JOB_HISTORY USING
   J.* = job_log
END_STORE
```

9.2.6.1.1 Using the CREATE_SEGMENTED_STRING Statement to Store Segmented Strings

Use the CREATE_SEGMENTED_STRING statement and the STORE statement to store segmented strings in a relation. You must use two operations when you store segmented strings. When you store a segmented string field, you are not actually storing the individual segments that comprise the segmented string into the field. The value that is stored in the segmented string field is an identifier, or logical pointer, to the location of the actual segmented string.

Note *The CREATE_SEGMENTED_STRING statement is available only in RDBPRE. To store a segmented string in RDML programs, use the STORE statement with segmented strings.*

First, use the CREATE_SEGMENTED_STRING statement to form the inner string of segments. Store the segments in this inner string with the STORE statement. Your program must explicitly repeat the STORE statement to store each segment, or iterate the STORE statement by a program loop. You cannot selectively store individual segments and you must store the segmented string in its entirety. For example, if you attempted to store first segment-1, next segment-3, next segment-5, and finally segment-2, your segmented string would contain: segment-1, segment-3, segment-5, and segment-2, in that order.

When all the segments are stored in a segmented string, use an outer STORE statement to store the segmented string identifier in a relation. (You can store other fields in the relation with the same STORE statement.) Once the outer STORE operation is complete, close the segmented string with the END_SEGMENTED_STRING statement.

You can close the segmented string before you perform the outer store operation that stores the segmented string identifier in a relation. However, do not use that segmented string identifier again until you have stored it in a relation.

The following pseudocode shows the logic involved in transferring a resume from a sequential file to a segmented string field in the sample personnel database. The uppercase statements are data manipulation statements. The lowercase text indicates the logic you must code in your host language. This example:

- Starts an outer loop with the CREATE_SEGMENTED_STRING statement. (Resume_handle represents the name you give to the segmented string stream.)
- Opens a file from which to read the resume information.
- Starts a loop that will be terminated at the end of the file that contains the resume.
- Reads a line from this file.
- Stores this line in the segmented string field of the RESUMES relation.

- Continues to read and store each line of the resume until the loop condition (end of file) is met.
- Closes the file that contained the resume.
- Stores the entire resume record in the database.

```
CREATE_SEGMENTED_STRING resume_handle
  open file from which you want to read resume
  information

  loop until end of file
    read line of file
    STORE R IN resume_handle USING
         R.RDB$VALUE = resume_line
    END_STORE
  end_loop

  if end_of_file then close file

END_SEGMENTED_STRING resume_handle

STORE R IN RESUMES USING
    R.EMPLOYEE_ID = employee_id;
    R.RESUME = resume_handle;
END_STORE
COMMIT
```

9.2.6.1.2 Using the STORE Statement with Segmented Strings to Store Segment Streams In RDML, use the STORE statement with segmented strings to create and store a segmented string field. Storing a segmented string involves two store operations; one STORE statement embedded within another.

Use the outer STORE statement to store all the fields in the record that are not SEGMENTED STRING data type. Specify a context variable and the relation into which you want to store all the fields in the outer STORE statement. Specify a second context variable and the field name (qualified by the context variable used in the outer STORE statement) in the inner STORE statement (the STORE statement with segmented strings). Use the inner STORE statement to store the segments.

RDML defines a special name to refer to the segments of a segmented string. This value expression is equivalent to the field name; it names the “fields” or segments of a segmented string. Furthermore, because the segments can vary in length, RDML also defines a name for the length of the segment. You must use these value expressions to retrieve or store the length and value of a segment. These names are:

- RDB$VALUE or VALUE

  The value stored in a segment of a segmented string.

- RDB$LENGTH or LENGTH

  The length of a segment in bytes.

When using the RDML and RDBPRE precompilers, be sure to define a sufficiently large value for the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name. An adequate buffer size is needed to store large segmented strings (using segmented string storage maps), in storage areas other than the default RDB$SYSTEM storage area. The minimum acceptable value for the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name must be equal to the sum of the length of the segments of the segmented string. For example, if you know that the sum of the length of the segments is one megabyte, then 1,048,576 bytes is an acceptable value for this logical name.

You must specify the logical name value because when RDML and RDBPRE precompilers store segmented strings, Rdb/VMS does not know which table contains the string until after the entire string is stored. Rdb/VMS buffers the entire segmented string, if possible, and does not store it until the STORE statement executes.

If the segmented string remains buffered, it is stored in the appropriate storage area. If the string is not buffered (because it is larger than the defined value for the logical name or the default value of 10,000 bytes), it is not stored in the default storage area and the following exception message is displayed:

```
%RDB-F-IMP_EXC, facility-specific limit exceeded
-RDMS-E-SEGSTR_AREA_INC, segmented string was stored incorrectly
```

To avoid this error, set the value of the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name to a sufficiently large value. Note that a value of up to 500 MB can be specified for this logical name. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for more information on defining storage areas.

Note *The SQL interface for lists (segmented strings) does not require you to define the value for this logical name. Before the list is brought into the buffer, SQL knows the column that the list is associated with and the table it is stored in. However, for large lists, defining this logical name with a value large enough to hold the entire list may improve the handling performance of storing the list.*

If the relation into which you are storing a record contains two segmented string fields, you must use two STORE statements with segmented strings in a series, within the outer STORE statement.

The following pseudocode demonstrates how to store a segmented string field using RDML. See Chapter 17 and Chapter 18 for RDML/C and RDML/Pascal examples. All lowercase words are host language variables. All uppercase statements are data manipulation statements.

```
open file from which you want to read resume
information

 STORE R IN RESUMES USING
    R.EMPLOYEE_ID = 12345

      loop until end of file
         read line of file
         STORE RR IN R.RESUME USING
           RR.VALUE  = first_line_of_resume
           RR.LENGTH = length_of_segment
         END_STORE
      end_of_loop

 END_STORE
```

9.2.6.2 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of one, many, or all the records in one relation. When you list the fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Use the FOR statement when you want to modify all the records in your record stream. You can take advantage of the automatic iteration of the FOR loop without having to use a program loop to step through the records. Use the START_STREAM statement when you want to conditionally modify the records in the record stream. You can use host language variables within your RSE so that your program logic can alter a record stream for each new FOR or START_STREAM statement.

The following example modifies the record of the employee with an employee ID that matches the value stored in the host language variable, employee_id. All lowercase words are host language variables. All uppercase statements are data manipulation statements.

```
START_TRANSACTION READ_WRITE RESERVING
   EMPLOYEES FOR SHARED WRITE
     FOR E IN EMPLOYEES WITH
       E.EMPLOYEE_ID = employee_id
          MODIFY E USING
             E.ADDRESS_DATA_1 = address_data_1;
             E.ADDRESS_DATA_2 = address_data_2;
             E.CITY = city;
             E.STATE = state;
             E.POSTAL_CODE = postal_code;
          END_MODIFY
     END_FOR
COMMIT
```

A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. You can copy definitions from the data dictionary to create such a record structure. (See Chapter 12 and Chapter 16 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement. The following example replaces the field values of an employee record in the JOB_HISTORY relation with the field values in the JOB_HISTORY record structure:

```
FOR J IN JOB_HISTORY WITH
   J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
   AND J.JOB_END MISSING
      MODIFY J USING
         J.* = JOB_HISTORY
      END_MODIFY
END_FOR
```

9.2.6.2.1 Modifying Segmented Strings in RDBPRE You can modify a record that contains a segmented string field, but you cannot modify the individual segments that comprise the segmented string field. The methods used to modify a segmented string differ for RDML and RDBPRE.

In RDBPRE, create a segmented string handle with the CREATE_SEGMENTED_STRING and STORE statements. Then modify the record that contains the segmented string with a MODIFY statement. Supply the resume_id name used in the CREATE_SEGMENTED_STRING statement as the new value for the segmented string field in the MODIFY statement. An example of this statement follows.

```
CREATE_SEGMENTED_STRING resume_id
    STORE R IN resume_id USING
       R.RDB$VALUE = resume_line
    END_STORE
END_SEGMENTED_STRING resume_id

FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id
   MODIFY R USING
     R.RESUME = resume_id
   END_MODIFY
END_FOR
```

9.2.6.2.2 Modifying Segmented Strings in RDML

You can modify a record that contains a segmented string field, but you cannot modify the individual segments that comprise the segmented string field. The methods used to modify a segmented string differ for RDML and RDBPRE.

In RDML, use a STORE statement with segmented strings within a MODIFY statement to change the value of a segmented string field. For example:

```
FOR R IN RESUMES
  WITH R.EMPLOYEE_ID = employee_id
     MODIFY R USING
        STORE RR IN R.RESUME USING
          RR.VALUE  = first_line_of_resume
          RR.LENGTH = length_of_segment
        END_STORE
     END_MODIFY;
END_FOR;
```

9.2.6.3 Erasing Records

You can delete one, many, or all the records from a relation using a single ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase. Depending on how you form your RSE, you can erase many or all the records from a relation with a single ERASE statement embedded in a FOR statement or within a stream formed by a START_ STREAM statement.

The ERASE statement can be an extremely expensive operation, using almost as many system resources as a load operation. In shared and protected share modes, each record erased generates a record in both the recovery-unit journal and the after-image journal. Thus, large-scale erasing of database records may exceed the enqueue limit (ENQLM). See the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for information on modifying system resources.

Use the FOR statement when you want to erase all the records in your record stream. You can take advantage of the automatic iteration of the FOR loop without having to use a program loop to step through the records. Use the START_STREAM statement when you want more control over erasing the records in the record stream. For example, when you want to branch to different procedures to perform stream manipulation tasks.

You can use host language variables within your RSE so that your program logic can alter a record stream for each new FOR or START_STREAM statement.

You can erase records from more than one relation by forming a record stream with a CROSS clause that joins several relations. However, if you use the ERASE statement with a CROSS clause that joins one record to many, you must be careful to erase only unique records.

For example, when you cross the EMPLOYEES relation with the DEPARTMENTS relation over the DEPARTMENT_CODE field, you form a record stream that contains one record for each employee ID. However, because a number of employees work in each department, this record stream contains a number of records that include the same department code field.

You can erase all the records in this stream using a context variable that points to unique EMPLOYEES records. However, if you want to erase the records in this stream using a context variable that points to the multiple DEPARTMENTS records, you must first use the REDUCED TO clause to make the DEPARTMENTS records unique. The following ERASE statement uses the context variable E to erase each record in the stream:

```
FOR D IN DEPARTMENTS CROSS E IN EMPLOYEES OVER DEPARTMENT_CODE
    ERASE E
END_FOR
```

To erase records in this record stream using the context variable D, you must first use the REDUCED TO clause to reduce the records in the stream so only unique DEPARTMENT_CODE fields will be included in the stream. If you attempt to erase multiple DEPARTMENTS records, the query will fail with the error RDMS-F-NODBK. The following RSE correctly uses the REDUCED TO clause to reduce the stream to only those records that contain unique DEPARTMENT_CODE field values before the stream is erased:

```
FOR D IN DEPARTMENTS CROSS E IN EMPLOYEES OVER DEPARTMENT_CODE
        REDUCED TO D.DEPARTMENT_CODE
    ERASE D
END_FOR
```

If you want to erase all department code records that contain duplicate values in the record stream, you should use nested FOR statements to create two separate streams for the DEPARTMENTS and EMPLOYEES relations. The following example erases the department FOO and all associated employees in that department.

```
FOR D IN DEPARTMENTS WITH D.DEPARTMENT_CODE = 'FOO'
    FOR E IN EMPLOYEES
      WITH D.DEPARTMENT_CODE = E.DEPARTMENT_CODE
          ERASE E
    END_FOR
      ERASE D
END_FOR
```

The following example demonstrates how you might erase a record from the sample personnel database. Because a record from the EMPLOYEES relation is being erased, it is a good idea to erase all the records that refer to the same employee ID, namely, those records in the JOB_HISTORY, SALARY_HISTORY, DEGREES, and RESUMES relations. Not only is this a good idea, but in the case of the sample personnel database, JOB_HISTORY and SALARY_HISTORY records for this employee must be deleted when the EMPLOYEES record that has the same value for EMPLOYEE_ID is deleted. If they are not, an error will be returned that indicates that constraints have been violated. The JOB_HISTORY and SALARY_HISTORY relations are defined with constraints that specify that an EMPLOYEES record must exist before an associated record (one with the same EMPLOYEE_ID field value) can be stored in either of these relations.

```
START_TRANSACTION READ_WRITE RESERVING
   EMPLOYEES, SALARY_HISTORY, JOB_HISTORY,
   DEPARTMENTS, DEGREES,
   RESUMES FOR SHARED WRITE
      FOR E IN EMPLOYEES WITH
         E.EMPLOYEE_ID  = "00167"

            FOR JH in JOB_HISTORY WITH
               JH.EMPLOYEE_ID = e.employee_id
               ERASE JH
            END_FOR

            FOR SH IN SALARY_HISTORY WITH
               SH.EMPLOYEE_ID = e.employee_id
               ERASE SH
            END_FOR

            FOR D IN DEGREES WITH
               D.EMPLOYEE_ID = e.employee_id
               ERASE D
             END_FOR

            FOR R IN RESUMES WITH
               R.EMPLOYEE_ID = e.employee_id
               ERASE R
            END_FOR
         ERASE E
      END_FOR
COMMIT
```

The next example performs the same erase operations as in the previous example. However, there would have to be a trigger definition as part of the database metadata that would implement a cascading delete. The cascading delete is triggered by the deletion of an EMPLOYEES record:

```
START_TRANSACTION READ_WRITE RESERVING
   EMPLOYEES FOR SHARED WRITE
      FOR E IN EMPLOYEES WITH
         E.EMPLOYEE_ID  = "00167"
         ERASE E
      END_FOR
COMMIT
```

The trigger definition that enables this cascading delete is as follows:

```
DEFINE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
        BEFORE ERASE
        FOR E IN EMPLOYEES EXECUTE
        FOR D IN DEGREES WITH
          D.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE D
        END_FOR;
        FOR JH IN JOB_HISTORY WITH
          JH.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE JH
        END_FOR;
        FOR R IN RESUMES WITH
          R.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE R
        END_FOR;
        FOR SH IN SALARY_HISTORY WITH
          SH.EMPLOYEE_ID = E.EMPLOYEE_ID
          ERASE SH
        END_FOR.
```

9.2.7 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer or address that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this dbkey as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, a dbkey is valid until you commit your transaction. That is, a dbkey is guaranteed to point to the same record only for the life of the transaction in which it is retrieved. In this case, the dbkey scope ends with the COMMIT statement.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

See the description of these statements in the *VAX Rdb/VMS RDO and RMU Reference Manual* and *RDML Reference Manual* for details on the syntax of the DATABASE statement.

One way of taking advantage of the dbkey scope qualifier is when you want to perform a large modify or erase operation. If you set the dbkey scope to end with the FINISH statement, you can start a read-only transaction to retrieve the dbkey for the subset of database records in which you are interested. You can then commit this transaction and begin a read/write transaction to perform the updates.

Because you have set the dbkey scope to end with the FINISH statement, you can be assured that the dbkeys you have retrieved will point to the same record they pointed to during the read-only transaction. However, *another user can modify or erase the records to which these dbkeys point.* Also, if you use this technique, be aware that other users may store records between the time you commit the read-only transaction and begin the read/write transaction. Before you use this technique, be certain that these issues do not affect the logical integrity of your data for your application.

If you can use this technique, you may find that by locating each record you want to update with the dbkeys retrieved in the read-only transaction, you do not subject the database to an excessive amount of locking. Dbkeys provide direct access to a record. If you do not use a dbkey to locate a record, Rdb/VMS might have to search through a relation or index to find the desired record. This search process may prohibit other users from accessing records involved in the search, until you have committed your transaction. For more information on locking see the *VAX Rdb/VMS Guide to Database Maintenance and Performance*.

Another effective use of dbkeys is to obtain the dbkey of a record that a program has just stored by placing a GET . . . RDB$DB_KEY expression in a STORE . . . END_STORE block. In this way, while the dbkey value is still valid, the program can make subsequent queries and use this value to access the record directly (instead of having to go through a search process for the record just stored).

9.2.8 Using Transactions

A **transaction** is an operation on the database that must complete as a unit or it will not complete at all. A transaction is bordered by a set of statements that begin with the START_TRANSACTION statement and end with either the ROLLBACK or COMMIT statement.

Between these borders can be any number of Rdb/VMS and host language statements. Transactions help to ensure that your application never partially updates a database. If your program terminates unexpectedly, active transactions are rolled back automatically by the database monitor. If you have designed your transactions properly, unexpected program termination will not leave the database in an inconsistent state.

Transactions are the core of a consistent multi-user, high-contention data processing environment. Often, a user insists that the state of the database remain unchanged (a high level of consistency) during the time the application runs. For example, when a ticket agent accesses a database and finds only one ticket left for tomorrow's flight to New York, that agent wants a guarantee that another agent cannot sell the ticket. Such a guarantee requires that the first application deny write access (the ability to sell and erase that last ticket) to the second application. In fact, the first application may not even want to grant read access to the second application. Such a high level of resource locking, while at times absolutely necessary, forces other concurrent applications to wait until the current transaction completes.

Applications in a much less restricted multi-user environment might be concerned only with what the database looks like at a particular point in time. When you start a transaction with the READ_ONLY qualifier, you access a *snapshot* of the database. No matter what updates other users might perform, you still see the state of the data at the time your transaction began. Thus, you can read while other transactions write, with minimum locking conflict.

Note *Rdb/VMS provides a transaction environment for applications running at a high level of consistency as well as for applications running at a high level of concurrency. See Chapter 2 for a full discussion of transactions.*

The length of a transaction may affect both the performance of the application and the consistency of the data in the database. In high-contention, interactive situations, you should strive for short transactions that lock the least number of records. On the other hand, each transaction must include all the data manipulation operations required to complete an update. For example, if a transaction updates a field, it must update that field in all the relations where the field exists. Otherwise, unexpected program termination could result in some of the values you intended to change remaining the same.

You should explicitly start a transaction before you execute any Rdb/VMS data manipulation statement. However, when a program module calls a submodule that includes data manipulation statements, you may not need to start a new transaction in the submodule. Before Rdb/VMS executes the submodule statements at run time, it checks to see if there is an active transaction of the appropriate type. If there is, the submodule data manipulation statements execute normally. If there is no active transaction, or if the transaction is read-only and the data manipulation statements perform data update, you will receive a run-time error.

Note *By default in RDML, and always in RDBPRE, if you do not explicitly start a transaction, or if you do explicitly start a transaction but do not specify the type of transaction you want, a read-only transaction is started for you. Furthermore, if your program accesses several databases and you do not specify the ON clause in a START_TRANSACTION statement, your program will attach to all the databases named in DATABASE statements within your program.*

Refer to Section 11.2.2 for information on how to override the default in RDML using the /NODEFAULT_TRANSACTIONS qualifier.

Language-specific examples of using the START_TRANSACTION statement are in the language-specific chapters.

9.3 Using Structured Programming in Preprocessed Programs

You should use structured programming concepts when you design your RDML and RDBPRE programs. Calls to routines or calls to subprograms and subroutines require special attention to:

- The use of context variables
- The DATABASE statement
- The use of transactions
- The START_STREAM statement

This next section discusses the scope of context variables in program blocks, and the use of the DATABASE statement and transactions in functions, subroutines, and submodules.

9.3.1 Using Context Variables in Program Blocks

Programs and modules that pass through one of the Rdb/VMS preprocessors, RDBPRE or RDML, do not have unlimited freedom in structure. Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- DECLARE_STREAM
- FOR
- GET
- START_STREAM
- END_STREAM
- FETCH
- STORE
- MODIFY
- ERASE
- CREATE_SEGMENTED_STRING (available only in RDBPRE)
- START_SEGMENTED_STRING (available only in RDBPRE)
- END_SEGMENTED_STRING (available only in RDBPRE)

These individual data manipulation statements each form only part of a complex call to the database. The preprocessor may generate one call to the database using more than one data manipulation statement. For example, a MODIFY statement executes within the context of a stream created by a FOR or START_STREAM statement. The call to the database can only be made using both the FOR and MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. In general, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

However, the declared START_STREAM statement lets you place the stream manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

Remember that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which the statement is issued.

Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- READY
- START_TRANSACTION
- END_STREAM (only when used with a declared stream)
- FETCH (only when used with a declared stream)
- GET (with statistical expressions or when used with a declared stream)
- COMMIT
- ROLLBACK
- FINISH
- Declared START_STREAM

Remember that you must issue the DECLARE_STREAM statement before you issue a declared START_STREAM statement and the DATABASE statement must appear in the data declaration section of your program.

9.3.2 Using Transactions in Separately Preprocessed Modules

If you treat each module as one or several separate transactions, start and end each transaction as you would normally. However, at times, program logic requires that a module execute in the context of a transaction that was started in another module.

For example, suppose your program uses two modules, MOD1 and MOD2, and MOD1 calls MOD2. If you start one transaction in MOD1 and a separate transaction in MOD2, you first have to end the transaction in MOD1 before you can call MOD2. Rdb/VMS permits only one active transaction per database attach. But suppose you do not want to commit the transaction in MOD1, and if you roll it back, you will lose the database values MOD2 intends to print.

This is not a problem because the default transaction handle is global to all the modules called by the program that starts the transaction. In other words, if you start a transaction in MOD1 and then call MOD2, the transaction started in MOD1 is available to MOD2.

In this scenario, do not use a START_TRANSACTION statement in MOD2. If you do not use transaction handles in the DML code used in MOD2, the default transaction handle will be available to MOD2. As your single executable image runs, Rdb/VMS will start a transaction in MOD1. When MOD1 calls MOD2, the transaction started in MOD1 is still open, and if the transaction declared in MOD1 allows the required operations in MOD2, Rdb/VMS will execute the statements in MOD2. If the transaction in MOD1 is not active, or does not permit the execution of the statements in MOD2, you will receive a run-time error.

9.3.3 Using Handles in Structured Programming

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions
- Requests

If you do not supply an identifier for these database objects, Rdb/VMS assigns default identifiers for you. In general, unless you need to make explicit references to one of the preceding database objects, it is not necessary for you to supply identifiers. Programmer-supplied handles are most often necessary when you attach to multiple databases (or attach to the same database more than once) within the same program.

As mentioned in the preceding paragraph, more than one handle can refer to the same object; for example, you can invoke the MF_PERSONNEL database twice within the same program and specify the database handle as "mypers" on the first DATABASE statement and as "pers" on the second DATABASE statement. Even though each handle is referring to the same physical database, Rdb/VMS will treat each attach to the database independently.

Note *The values of request handles and transaction handles must be zero the first time your program refers to a particular object. If any of these handles is not zero the first time your program refers to a particular object, Rdb/VMS will return an error that indicates that the handle was bad. If your program changes the assigned value in any way, later attempts to use that handle will generate the same error.*

RDBPRE and RDML supply two qualifiers: /INITIALIZE_HANDLES, the default, and /NOINITIALIZE_HANDLES. These qualifiers let you determine whether or not the preprocessor will automatically initialize database, transaction, and request handles. These qualifiers have no effect on if or when handles are *initialized* in the generated code; they only control initialization of handles in declaration statements. Furthermore, they only affect database, transaction, and request handles that the preprocessor declares; user-specified transaction and request handles will not be initialized when you use the /INITIALIZE_HANDLES qualifier.

RDML will initialize database handles supplied by the user and by RDML when their scope is GLOBAL or LOCAL. Database handles with EXTERNAL scope are never initialized. For details on how to specify these qualifiers on the RDML command line see Chapter 11.

Note *In RDBPRE when you use the /NOINITIALIZE_HANDLES qualifier, any handle you specify in your application program must also be specified in the shareable image if your application is built using a shareable image.*

9.3.3.1 Using Database Handles A **database handle** is a variable name you use to refer to a database. Database handles are used to distinguish between two different active databases referred to in the same program or to distinguish between two different attachments to the *same* database. You can refer to more than one database in any single RSE; however, you cannot cross relations from different database attaches.

If you specify a database handle explicitly, you do so in the DATABASE statement. You can then use the database handle in several statements and clauses in order to identify the database which you are accessing, among them:

- CREATE_SEGMENTED_STRING (RDBPRE statement)
- FINISH
- START_TRANSACTION

- Any RSE
- READY

In addition to specifying database handles, you can specify the scope of database handles in RDBPRE and RDML programs. (You cannot specify the scope of database handles in Callable RDO programs or statements.) The scope of a database handle can be GLOBAL, LOCAL, or EXTERNAL. If you do not explicitly state the scope of the database handle, both RDML and RDBPRE default to a GLOBAL scope. If your RDML or RDBPRE program invokes multiple databases, use database handles in each DATABASE statement.

When you access multiple databases and your program includes multiple modules:

- Include in the main module DATABASE statement a database handle for each database accessed.
  - Use the GLOBAL database handle scope for any database handle that is also invoked in a submodule.
  - Use the LOCAL database handle scope for any database handle that is only invoked in the main module.
- Include in each submodule DATABASE statement a database handle for the database accessed.
  - Use identical database handles for the same database attachment.
  - Use the EXTERNAL database handle scope for any database handle that resides outside the submodule.
  - Use the LOCAL database handle scope for any database handle that is only accessed in the submodule.

Your format for the DATABASE statement depends on how many databases you access and how many separately preprocessed modules make up your program image. If you use subprogram or function modules that access the same database as the main module (the first module that invokes the database), you must:

- Include database handles
- Include a database handle scope in preprocessed programs
- Not include a database handle scope in Callable RDO programs

When Rdb/VMS performs the DATABASE statement, it returns a value to the database handle. Rdb/VMS will use the same value for each time the database is invoked with the same database handle.

Table 9–1 summarizes the use of database handles in RDBPRE and RDML preprocessed programs. Table 9–2 summarizes the use of database handles in Callable RDO programs.

Table 9–1 Summary of Database Handle Usage in RDML and RDBPRE Preprocessed Programs

Number of Databases	Number of Modules	Handle Scope in Main Module	Handle Scope in Second Module	Handle Scope in Additional Modules
One	One	Not required	Not applicable	Not applicable
One	Multiple	GLOBAL	EXTERNAL	EXTERNAL
One	Multiple	EXTERNAL	GLOBAL	EXTERNAL
Multiple	One	LOCAL	Not applicable	Not applicable
Multiple	Multiple	GLOBAL	EXTERNAL	EXTERNAL
Multiple	Multiple	EXTERNAL	GLOBAL	EXTERNAL

Table 9–2 Summary of Database Handle Usage in Callable RDO Programs

Number of Databases	Number of Submodules	Database Handles Required?
One	None	No
One	One or multiple	No
Multiple	None	Yes
Multiple	One or multiple	Yes

9.3.3.2 Using Transaction Handles A **transaction handle** is a host language variable that identifies a particular transaction. If you do not declare the transaction handle explicitly, the preprocessor uses the default transaction handle.

Because Rdb/VMS permits only one active transaction per database attachment, it is usually not necessary to use transaction handles in Rdb/VMS programs.

However, you can attach to the same database twice and start a single transaction against each database attachment within the same program. This requires the use of both database and transaction handles. Details of how to use request handles are supplied in Section 9.3.3.3.

Information on how to declare transaction handles in a given language is contained in the language-specific chapters.

9.3.3.3 Using Request Handles

A **request handle** is an integer longword that points to the location of a compiled Rdb/VMS request. A request handle serves as a pointer to the internal representation of a query. By using a request handle, the preprocessors can cause the database system to use this internal representation again, thus reducing the overhead associated with repeatedly executing a query. *This happens whether you specify request handles explicitly or not.* You should consider the need to supply request handles as the exception rather than the rule. Usually, it is unnecessary for you to supply request handles. Incorrect use of user-supplied request handles can make your program needlessly complicated and difficult to debug. Specific guidelines of when you should and should not use request handles are in Section 9.3.3.3.1.

The following example demonstrates how a request handle is used by RDBPRE or RDML. Assume you write code in your host language program based on the following pseudocode:

```
define necessary variables and invoke database;

solicit COLLEGE_NAME value from user;

start read_only transaction;

FOR C IN COLLEGES WITH C.COLLEGE_NAME = host_variable
  print location of the college with the host_variable name
END_FOR

commit transaction
end program.
```

RDBPRE and RDML each create a request handle to identify the query for searching the database for a COLLEGE_NAME value equal to the value of the host language variable.

If the host language code for the preceding query were executed twice in the same program (with no intervening FINISH statement), the second execution of the query would *not* be compiled; Rdb/VMS would assign the request handle used in the first request to the “twin” request that occurs later in the program, thus saving the resources associated with compiling a request.

9.3.3.3.1 Determining When to Use User-Supplied Request Handles

In most instances you should not supply request handles, because they are unnecessary and add needlessly to the program’s complexity. In particular, do not supply request handle names if:

- You are accessing only one database (regardless of the number of modules) and you are not using a FINISH statement between calls to the modules. You can accept the handles that Rdb/VMS supplies by default with no loss in performance.
- You are using multiple databases and issuing separate queries for each database (each query refers to its own database using database handles).

There are two main reasons for you to supply request handles:

- You need explicit control over resource allocation.

  If you need to conserve virtual memory, you can use the RDB$RELEASE_REQUEST procedure to release the resources allocated to a request, as long as you are certain that your program will never use that query again. However, if you find that your program consumes large amounts of virtual memory, find out why this is happening. Improper use of user-supplied request handles can result in excessive use of virtual memory.

- You are using multiple modules that attach and reattach to the database and you want to use the request handles again.

  If you are using separately compiled modules, the submodules must use request handles and must provide a means for initializing them when the FINISH statement is executed (that is, before a new database attachment occurs).

  For example, suppose you have a main module named MYPROGRAM that issues a READY statement for the MF_PERSONNEL database. You also have a module named PROG2 that contains a database query. MYPROGRAM and PROG2 are separately processed modules.

  The following pseudocode demonstrates how you could use request handles to successfully call PROG2 from MYPROGRAM:

  MYPROGRAM:

```
DATABASE GLOBAL PERS = FILENAME MF_PERSONNEL
       .
       .
       .
READY PERS
       .
       .
       .
CALL PROG2
       .
       .
       .
FINISH PERS
       .
       .
       .
reinitialize request handle, req1, to zero
       .
       .
       .
READY PERS
CALL PROG2
       .
       .
       .
```

PROG2:

```
DATABASE EXTERNAL PERS = FILENAME MF_PERSONNEL
       .
       .
       .
FOR (REQUEST_HANDLE req1) E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = '00125'
       .
       .
       .
END_FOR;
```

Note that in the preceding example, the compiled request is *not* used again (even though the request *handle* and the associated query are.) The request is compiled again each time PROG2 is called. If you did not initialize the request handle to zero after the database PERS was finished, you would receive a RDB$_BAD_REQ_HANDLE error message. This is because an invalid request handle value would be accessed after the first call to PROG2.

9.3.3.3.2 Declaring and Initializing a Request Handle A request handle is created when:

- You embed DML statements to form a query in your program.

  In this case, RDBPRE and RDML translate your query into a request to the database, declare a unique name for this request, and *initialize the value of the name to zero*. In other words, the preprocessors create and use request handles regardless of whether you supply the name for them or not.

- You embed DML statements to form a query and supply your own request handle name.

  In this case, RDBPRE and RDML translate your query into a request to the database and assign a value to the name you supply for the request handle. However, you must initialize the request handle to zero before you refer to it in a query.

The preprocessors initialize the request handle or handles that they create before the handle is first used. When your program issues a FINISH statement:

- RDBPRE does not reinitialize any request handles compiled under the finished database.

  When a database is finished, Rdb/VMS releases the resources allocated to the compiled request and the value of the request handle becomes meaningless. Therefore, an attempt to finish a database, then attach to the same database and continue to use the request associated with the finished database, will result in a RDB$_BAD_REQ_HAND error at run time.

- RDML reinitializes to zero all request handles that it declares in the same module in which the FINISH statement appears. Therefore, an attempt to finish a database, then attach to the same database and continue to use the internal request associated with the finished database will work. However, the compiled request will not be used again. In this case, RDML will compile the query twice.

  When you supply the request handle name, you are responsible for declaring it and initializing the handle to zero before you use it and initializing it again after you issue a FINISH statement.

Note *By default, RDML initializes handles that it declares. However, when you specify the /NODEFAULT_TRANSACTIONS qualifier on the RDML command line, RDML does not initialize the handles it declares. See Chapter 11 for a discussion of how the /NODEFAULT_TRANSACTIONS qualifier affects RDML request handles.*

9.3.3.3.3 Changing the Value Associated with a Request Handle You should change the value of a user-supplied request handle only when you initialize it before it is first used, and initialize it again when it is no longer needed. You should never attempt to change the value of a request handle generated by RDML or RDBPRE.

When the code generated by a preprocessor encounters a request, it checks the value of the request handle. If this value is zero, the preprocessor compiles the request. If it is not zero, the preprocessor assumes the request has already been compiled and will use the compiled request again, as is required.

However, if you change the value of the request handle to zero and then attempt to use it again, you will be using the same variable associated with the request handle, but you will not be using the same compiled request. Your program will begin to unnecessarily consume system resources which may lead to a EXQUOTA error message. Do not change the value of a request handle to zero if you want to continue using the compiled request to which the request handle points.

If you change the value of the request handle to a number other than zero and then attempt to use it again, you may receive the error message RDB$_BAD_REQ_HANDLE, or you may get incorrect results, or a REQSYNC error because you may assign a value that refers to another request.

You can, (but should not), assign a non-zero value to a request handle explicitly by using a host language assignment statement.

You render a request handle invalid by issuing a FINISH statement that detaches from the database against which the request was compiled. Therefore if you need to use it again, you must initialize it again after the FINISH statement is issued.

9.3.3.3.4 Determining the Scope of a Request Handle The value of a request handle is valid from the point a query is first made until the request handle is explicitly released (with RDB$RELEASE_REQUEST), or until the database associated with that query is detached using a FINISH statement. When you release a request handle, you are releasing the compiled request and the resources associated with it, and thus reducing your use of virtual memory.

A request handle should be global to the routines that use it. If a request is compiled in a subroutine, the value of the request handle must be passed back to the calling module. If the value of the request handle is not passed back to the calling module, the query will be compiled again each time the subroutine is called and the resources associated with multiple compilations of the same query will not be released until the application detaches from the database. Note that applications that use request handles incorrectly usually run out of virtual memory after a while.

A given request handle is valid only when used with the database handle under which the request was compiled. This means that a compiled request can only be used to access one database attachment; it cannot be used to access multiple databases. An attempt to use a compiled request with a different database handle, even if it refers to a different attachment to the same database, will result in an error. A FINISH statement releases the resources allocated to a database handle; therefore, if you specify a request handle for a query and then issue a FINISH statement, the value of the request handle is no longer valid. For example, a program that is based on the following pseudocode will generate the RDB$_BAD_REQ_HANDLE error message:

```
invoke database
declare request handle and initialize to zero

ready
start transaction
    .
    .
    .
FOR (REQUEST_HANDLE handle1) E IN EMPLOYEES
  print EMPLOYEE_ID
END_FOR;
    .
    .
    .
COMMIT;
FINISH;

FOR (REQUEST_HANDLE handle1) E IN EMPLOYEES
  print EMPLOYEE_ID
END_FOR;
```

You will not receive an error message if you revise the preceding pseudocode such that the request handle is initialized to zero before it is used for the second query. However, this will result in the same query being compiled twice, unnecessarily.

```
invoke database
declare request handle and initialize to zero
start transaction
    .
    .
    .
FOR (REQUEST_HANDLE handle1) E IN EMPLOYEES
  print EMPLOYEE_ID
END_FOR;
    .
    .
    .
COMMIT;
FINISH;

reinitialize request handle to zero

FOR (REQUEST_HANDLE handle1) E IN EMPLOYEES
  print EMPLOYEE_ID
END_FOR;
```

9.3.4 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference. See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

9.4 Using Callable RDO in Preprocessed Programs

The RDBPRE and RDML preprocessors do not support data definition statements. If you want to perform data definition within your preprocessed program you must use the Callable RDO program interface. For example, during a batch job, or when no one else is using the database, your program may define a temporary index on a field to facilitate Rdb/VMS performance during your program execution. When using Callable RDO, your program communicates with Rdb/VMS using a callable function, named RDB$INTERPRET. Callable RDO program development is explained in detail in Chapter 19.

You can also use Callable RDO when your program needs the ability to form dynamic queries; that is, when your program will not know what a query is until run time. Otherwise, you should use the RDBPRE or RDML preprocessor when possible for all data manipulation operations. Preprocessed Rdb/VMS statements execute significantly faster than calls using the function RDB$INTERPRET.

This section discusses the use of the DATABASE statement and the scope of transactions in preprocessed programs that use Callable RDO.

9.4.1 Using the DATABASE Statement with Embedded Callable RDO

You must use a DATABASE statement in your preprocessed program and a separate RDO DATABASE statement in the embedded Callable RDO. To ensure that the preprocessor invokes the identical database for the preprocessed Callable RDO portions of the program, use the same database handle in each DATABASE statement. Invoke the database:

- In the preprocessed programs using a GLOBAL or EXTERNAL database handle.
- In Callable RDO programs, pass the database handle to the RDB$INTERPRET function.

In Callable RDO, you must pass the database handle to the RDB$INTERPRET function as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both preprocessed and Callable RDO DATABASE statements in the same program module. You may also call a function or subroutine to perform the data definition. In that case, use a preprocessed DATABASE statement in the main module and the Callable RDO DATABASE statement in the submodule.

9.4.2 Using Transactions with Embedded Callable RDO

Data definition statements require a read/write transaction. When an Rdb/VMS program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You must perform Callable RDO data definition statements within a read/write transaction. However, if you start a read/write transaction in the Callable RDO portion of your program, make sure that you commit or roll back any active transactions you started in the preprocessed portion of your program first. If a transaction is active in your program when you issue the START_ TRANSACTION statement with a Callable RDO statement, your Callable RDO statement will return a run-time RDO error.

Note that if you call the RDB$INTERPRET function for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view using Callable RDO within a preprocessed program and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist. The preprocessor will generate errors for those statements that refer to nonexistent fields, relations, or views. However, you can define indexes and constraints in Callable RDO statements that are embedded in preprocessed programs and any other database element that is not referred to in the preprocessed code.

You can perform any valid preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than to rely on implicit START_TRANSACTION declarations.

10 Handling Rdb/VMS Run-Time Errors in Preprocessed Programs

This chapter is a general description of how to detect and handle Rdb/VMS errors that occur at run time in preprocessed programs. This chapter discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your programming language user's guide for such information. Refer to the language-specific chapters for information on how to implement the error handling routines described in this chapter. Refer to Chapter 19 for specific information on error handling in Callable RDO.

10.1 Program Design and Error Handling

All programs are subject to run-time errors, both expected and unexpected. You can handle expected errors in a predictable manner. Some expected "errors" are not errors in the true sense but rather expected exception conditions. For example, when you use the START_STREAM statement to form a record stream, you can detect and handle the end-of-stream condition. In most instances, you want to detect expected errors using an error handler that tests the error and conditionally allows your program to continue execution. Some examples of expected errors are:

- Normal end of record stream and end of segmented string
- Deadlock and lock conflict
- Validation, constraint, and duplicate value violations

Other errors are unexpected in the sense that your program fails to detect and handle them. Frequently, unexpected errors are entirely unpredictable. In some instances, you want to display the error message and terminate the program so you can determine the cause of the error and correct it. Some examples of unexpected errors are:

- Access violations
- Arithmetic exceptions
- Invoking the wrong database
- Obsolete metadata
- Corrupt database

The distinction between expected and unexpected errors largely depends upon your application. You should anticipate and handle as many errors as is reasonable within the context of your application. Your program can recover from almost any error, even if it is fatal, with the proper program design. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation.

If you fail to include a handler for Rdb/VMS errors in your preprocessed program and an Rdb/VMS error occurs, the preprocessors generate code to call the VMS Run-Time Library routine, LIB$STOP. The LIB$STOP routine forces the severity level of the error to fatal, causing the program to terminate, and displays the error message on your terminal. In Callable RDO programs, if you do not handle an unsuccessful call, your program will continue in an unpredictable manner.

A well-designed program includes one or more error handlers that process Rdb/VMS errors in a predictable manner. These error handlers would:

- Detect all Rdb/VMS errors
- Handle all anticipated errors:
  - By error message display
  - By a recovery procedure or orderly program termination
- Handle all unanticipated errors:
  - By error message display
  - By orderly program termination

Regardless of the type of error you are handling, your error handlers should include one or more of the following functions:

- Error detection

- Error message display (user-supplied, Rdb/VMS, or a lower-level facility such as VMS)
- Error detection and conditional branching
- Error recovery

The error handling routines and strategies that you use in RDBPRE and RDML preprocessed languages differ from those you use in Callable RDO. In Callable RDO, every statement must be checked for successful execution with a conditional host language statement. In RDBPRE and RDML preprocessed languages, the ON ERROR clause of each DML statement checks the success of statement execution for you. Callable RDO and the preprocessed languages also differ in the manner in which they use system service and run-time library routines. For more information on handling errors in Callable RDO, see Chapter 19.

If you choose to combine Callable RDO and preprocessed Rdb, use separate error handling routines for Callable RDO and preprocessed DML statements.

10.2 Error Handling for Preprocessed Programs

Preprocessed programs allow you to detect Rdb/VMS errors with the ON ERROR clause. If an error occurs in a data manipulation statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- Detecting and displaying errors using the default VMS condition handler
- Detecting and handling errors using the ON ERROR clause
- Determining which error has occurred using symbolic error codes
- Displaying error messages using VMS system services and Run-Time Library routines
- Displaying user-supplied messages
- Error recovery

10.2.1 Detecting and Displaying Errors—Default Condition Handling

If you do not supply an error handler, Rdb/VMS uses the two VMS operating system default condition handlers: the traceback and catchall handlers. By default, the traceback handler is included when you link your program. Once you have completed program development, you generally link your program with the /NOTRACEBACK qualifier and use the catchall handler instead.

The traceback handler displays the message associated with the signaled condition code, the traceback message, the program unit name and line number of the statement that signaled the condition code, and the relative and absolute program counter values. (On a warning or error, the line number of the next statement to be executed is displayed.) In addition, the traceback handler displays the names of the program units in the calling hierarchy and the line numbers of the invocation statements. After displaying the error information, the traceback handler continues program execution or, if the error is severe, terminates program execution.

The catchall handler displays the message associated with the condition code and then continues program execution or, if the error is severe, terminates execution. The catchall handler is not invoked if the traceback handler is enabled.

For example, if you start a read-only transaction and attempt an update operation, the following text is displayed at run time (assuming you have linked your program without the /NOTRACEBACK qualifier).

```
%RDB-F-READ_ONLY_TRANS, attempt to update from a read_only transaction
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name                     line       rel PC    abs PC

                                                           00017940  00017940
RDML_SIGNAL_ERR RDML$SIGNAL_ERROR                 651      00000015  0000093E
MULTIPLY        MULTIPLY                          542      00000235  00000871
```

Although these messages are helpful when you are debugging your program, you probably want errors to be displayed and handled in a different manner when an end user runs your program.

10.2.2 Detecting and Handling Errors Using the ON ERROR Clause

Instead of having your application terminate as previously shown, you should use the ON ERROR clause to help your program to determine that an error detected by Rdb/VMS has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. You can use the ON ERROR clause only in preprocessed programs. All of the executable data manipulation statements offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you include one or more host language or DML statements or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

Note that the ON ERROR clause detects only Rdb/VMS errors. It does not detect other kinds of errors, such as access violations, divide-by-zero errors in the host language, and out-of-bounds arrays.

When using the ON ERROR clause for error detection, you do not have to detect an error explicitly. All communication with Rdb/VMS is done through procedure calls; the preprocessor generates a conditional statement that tests the return status value of each call to the database. When an error occurs, the return status indicates that the call failed. Control passes to the ON ERROR clause and the statements within the ON ERROR block are executed. If, on the other hand, the return status indicates success, the ON ERROR block is ignored.

Error handling with the ON ERROR clause may require special program design. When an error occurs and control passes to the ON ERROR clause, you may not want to continue processing the record stream in which the error occurred. For example, if you encounter an integrity failure during a modify operation, you must exit the record stream to handle the error. Do not attempt to handle the error within the ON ERROR clause and continue processing the record stream. You may call a separate procedure or function to handle the error from the ON ERROR block, but when you return to the ON ERROR block from the module, you should use an appropriate host language statement to exit the FOR loop. You can then start the query again if you choose.

You may use the ROLLBACK statement within an ON ERROR block. However, do not attempt to start the transaction again by using a START_TRANSACTION statement within the ON ERROR block. Instead, exit the record stream by using a host language GOTO statement in the ON ERROR block. Then issue the START_TRANSACTION statement.

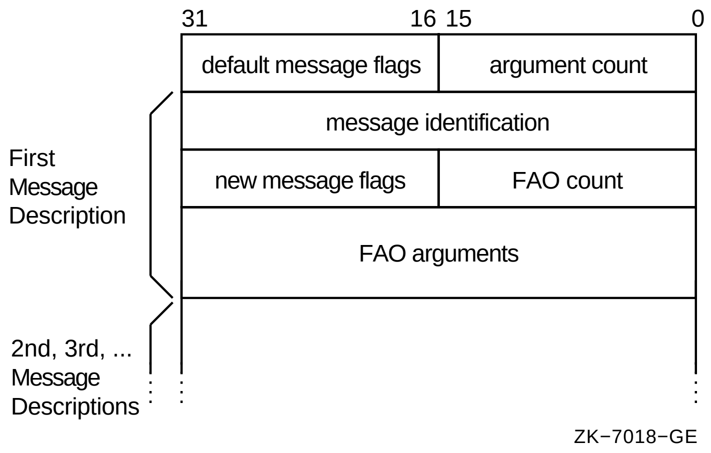
If you do not use the ON ERROR clause and an Rdb/VMS error occurs, the code generated by the preprocessors passes the error to LIB$STOP, which sets the severity level to fatal and forces program termination.

Language-specific restrictions and examples of using the ON ERROR clause appear in each language-specific chapter.

Once you have determined that an error has occurred by using the ON ERROR clause, you need to determine which error has occurred by examining the error code.

As previously stated, all communication with Rdb/VMS is done through procedure calls. In preprocessed programs, the preprocessor converts Rdb/VMS statements to host language calls to Rdb/VMS procedures. Every procedure returns a return status value to a program variable, RDB$STATUS, that is declared by the preprocessor. The return status value is a longword that identifies a unique message in the system message file. The message vector that contains RDB$STATUS is called RDB$MESSAGE_VECTOR. Figure 10–1 illustrates the format of the message vector.

Figure 10–1 The Format of the Rdb/VMS Message Vector



The return status value returns a condition that may indicate success, in which case data manipulation continues uninterrupted. Or this value may signal an error, in which case control passes to the error handler.

An Rdb/VMS symbolic error code is associated with each unique return status value. For example, RDB$_STREAM_EOF is the symbolic error code for the end-of-stream condition. See Table A–1 in Appendix A for a list of commonly used Rdb/VMS symbolic error codes for data manipulation statements. Table A–1 is not an exhaustive list; you might want to create a list of likely and less likely errors for your particular application or programming facility. The *VAX Rdb/VMS RDO and RMU Reference Manual* contains pointers to the online Rdb/VMS error message explanation files.

You can use these symbolic error codes to control program logic for specific errors. When the ON ERROR clause detects an error, your error handler can evaluate the symbolic error code by:

- Calling LIB$MATCH_COND, a VMS Run-Time Library routine that compares the signaled condition code to a list of expected condition codes (error codes).
- Using a local host language equality test, for example, a case statement or nested "if" statements that test to see if the signaled condition code matches one in a series of expected condition codes.

Then your error handler can direct program logic with a host language multipath statement, such as the Pascal CASE statement or the COBOL EVALUATE statement.

Although symbolic names, such as RDB$_DEADLOCK, symbolize actual values, you should use the symbolic names in your programs (rather than their value) for the following reasons:

- The symbolic error codes themselves are mnemonic. You can assign your own mnemonic names in some programming languages.
- The VMS Linker automatically assigns the numeric values to the symbolic names for you.
- If the numeric value of a symbolic error code ever changes, all you have to do is link your program again. If you have coded values into your program, you have to search for and change each occurrence of the value in the source files and preprocess the files again.

For information on declaring symbolic error codes and calling the LIB$MATCH_COND routine in preprocessed programs, see the language-specific chapters or your programming language user's guide.

10.2.3 Displaying Error Messages in Preprocessed Programs

The method you choose to determine which errors have occurred and to display error messages depends on several factors. If you want to:

- Determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine, LIB$MATCH_COND
- Display an error message generated by Rdb/VMS and (optionally) terminate your program, you can call the LIB$SIGNAL routine
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system services with a user-defined error code

For more information on how to make these calls, see the language-specific chapter for the host language you are using. If your host language is not supported by a preprocessor, see Chapter 19.

10.2.4 Displaying User-Supplied Error Messages in Preprocessed Programs

After detecting expected errors, you may want to display your own error messages. In this way you can format the messages as you wish and include more specific information about error recovery than that supplied by the Rdb/VMS error messages. To display user-supplied error messages, you may include literal error messages (error messages in quoted strings) within your program's error handler. However, Digital recommends that you use the VMS Message utility (MESSAGE) to define your own error messages. Your error handler can then use SYS$GETMSG and SYS$PUTMSG system services to retrieve and display these user-defined messages.

Create a source message file (file type MSG) with MESSAGE or use a text editor to create a standard file with the MESSAGE format. The following is a short (two-message) user-defined message file called COBOLMSG.MSG:

```
.TITLE ADMINMSG  'Sample Error Messages'
.FACILITY ADMSAMPLE,27/PREFIX=ADM_
.SEVERITY       FATAL
NODUEMP         <No duplicate employee ids permitted>
EMPLOCKED       <The employee record requested is currently locked>/FATAL
.END
```

When users attempt to store two employees with the same ID, they see the following error message:

```
%ADM-E-NODUPEMP, No duplicate employee ids permitted
```

To access this file from your program:

1 Compile the message source, COBOLMSG.MSG, by itself with the command:

```
$ MESSAGE/NOSYMBOLS COBOLMSG
```

2 Create a nonexecutable message file by linking the object module, COBOLMSG.OBJ, with itself. For example:

```
$ LINK/SHAREABLE = COBOLMF COBOLMSG.OBJ
```

3 Create a pointer object module, MESPNTR.OBJ, that will point to the nonexecutable message file, COBOLMF.EXE. The /NOTEXT qualifier indicates that what follows contains only the symbolic error codes and no text. Use the command:

```
$ MESSAGE/FILE_NAME=COBOLMF /NOTEXT/OBJECT=MESPNTR COBOLMSG
```

4 Link the pointer object module, MESPNTR.OBJ, with the COBOL program object module, COBOLCODE.OBJ, with the command:

```
$ LINK COBOLCODE, MESPNTR
```

5 Execute the program with the command:

```
$ RUN COBOLCODE
```

Once a reference to the message file is created in the program's executable image, you can edit and link the message text and pointer files again without linking the program again.

Note that if you customize Rdb/VMS error messages to create new messages for your application, you should not retrieve an RDB, RDMS, or RDO message text string (with Formatted ASCII Output (FAO) arguments included) and then parse the characters in the text string to locate and retrieve the substitution value for the FAO arguments if the following is true:

- You are creating an application that retrieves information returned by the VMS message vector for error conditions.
- Your application returns messages customized for the user of your application.
- Your application's message text includes FAO arguments included in RDB, RDMS, and RDO facility error messages.

Message text supplied by Digital products is subject to change from one version of a product to another. Text changes may be made to improve message clarity or make corrections to text that is misleading. If your application parses RDB, RDMS, or RDO facility message text (for example, expecting the number of characters before and between FAO substitution values to remain constant), a future release of Rdb/VMS software may cause your application to fail.

To retrieve formatted values for FAO arguments to be used in text that you supply, follow the instructions provided in the system routines volume of the VMS documentation set. These instructions explain the strategy supported by Digital products that use VMS message vectors.

Please note that to ensure compatibility of your application from one version of Rdb/VMS to a new one, any future changes to RDB, RDMS, or RDO message text will not affect:

- The ordinal position of error messages in the message files
- The number of FAO substitution arguments embedded in the text of a message
- The order in which FAO substitution arguments occur in the text of a message

10.2.5 Recovering from Errors in Preprocessed Programs

Error recovery is specific to the program in which the error occurs. Frequently, the individual program logic requires an individual error routine. However, there are several categories of Rdb/VMS errors that preprocessed programs share in common:

- Multi-user conflicts

  Deadlock and lock conflict are likely to occur in high-contention, multi-user applications, particularly those that access the database in protected or exclusive share modes. In single-user databases, or databases that are accessed solely by using the shared read-only reserving option, deadlock and lock conflict are rare, but not impossible, occurrences.

- Integrity and constraint failures

  If your database has requirements for unique indexes, or validity and constraint checking, program input data may be rejected. In an interactive program, invalid input is normally handled by rolling back the transaction, requesting valid data from the user, and trying again. Programs that are not interactive might write the error to a file and continue with the next input record. The exact strategy for handling invalid input depends upon the individual logic of the program.

- Fatal or unexpected errors

  Some fatal errors are entirely unanticipated, and therefore difficult to handle during the execution of the program. In this case, you may choose to display an error message and terminate the program in an orderly manner.

  Other fatal errors can be anticipated. In this case, you may wish to direct program logic to continue program operation after an anticipated fatal error occurs. To handle this case, your error handler can call a VMS Run-Time Library routine or a system service to display the error message, and then allow the program to branch to alternative program code.

10.2.5.1 Handling Multi-User Conflicts in Preprocessed Programs If your program runs in an environment where other users are accessing the database using protected or exclusive read or write reserving options, you can encounter deadlock and lock-conflict error conditions. Lock-conflict and deadlock conditions are described as follows:

- Lock conflict occurs when you have specified the NOWAIT option in your START_TRANSACTION statement and the data your program needs is locked by another user's protected or exclusive transaction. Lock conflict can also occur in some read-only transactions, even if the WAIT option is specified.

Your program should detect and handle lock conflict even if you specify the default, WAIT, in your START_TRANSACTION statement. Rdb/VMS considers the exclusive share mode and the read-only lock type to be incompatible. If a user has started a transaction in the exclusive share mode, and you attempt to open a read-only transaction with the wait option, Rdb/VMS converts your transaction to nowait and issues a lock-conflict error to your program. Thus, it is good practice to anticipate lock conflict even when it seems extremely remote.

- Deadlock occurs when the data you need is locked by another user and you have locked the data the other user needs; neither of you can continue.

  Deadlock is a relatively rare situation, but when it occurs, the VMS Lock Manager picks one user and returns the deadlock error condition to that user. (See the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for information on lock management.) If you do not detect and handle that condition, your program will terminate abruptly.

To handle multi-user conflicts in preprocessed programs:

- Detect deadlock by looking for the RDB$_DEADLOCK error code
- Detect lock conflict by looking for the RDB$_LOCK_CONFLICT error code

When you detect a lock-conflict condition, you have the choice of rolling back the transaction and:

- Trying again one or more times
- Trying again after a set period of time
- Starting the transaction again and specifying the WAIT option
- Terminating the program and running it at another time

When you detect a deadlock condition, you have the choice of rolling back the transaction and:

- Trying again one or more times
- Trying again after a set period of time
- Terminating the program and running it at another time

Note that the START_TRANSACTION, ROLLBACK, and FINISH statements require a differently designed ON ERROR clause. If a START_TRANSACTION statement returns a deadlock or lock conflict, you cannot roll back because there is no active transaction. Simply attempt to start the transaction again in one of the ways previously listed. The ROLLBACK and FINISH statements do not encounter deadlock and lock-conflict conditions, and so are not rolled back.

10.2.5.2 Handling Integrity Failures in Preprocessed Programs

Integrity failures are caused by the following:

- A constraint violation

  The DEFINE CONSTRAINT statement sets conditions that restrict the values stored in a relation. For example, a constraint can require that a department must exist before the corresponding department code can be stored in the JOB_HISTORY relation.

- A violation of the VALID IF clause

  The VALID IF clause in the DEFINE FIELD and CHANGE FIELD statements sets conditions that restrict the values stored in a field. For example, a VALID IF clause can require that an employee ID lie within a certain range of values.

- A violation of the DUPLICATES ARE NOT ALLOWED clause

  The DUPLICATES ARE NOT ALLOWED clause in the DEFINE INDEX statement requires that each value in the index be unique.

The way that you handle integrity and constraint failures depends on whether your program is interactive or is running in batch mode. If your program is:

- In interactive mode

  Your error handler typically displays an error message that indicates the type of failure and how to correct it. Your program rolls back the transaction, requests new input, and issues the START_TRANSACTION statement again.

  When you design interactive data input programs, consider the amount of data that will have to be entered again if the program must be rolled back. It is a good idea to nest the START_TRANSACTION . . . COMMIT statement block within the data input loop, so that in the event of a rollback, only the last data item will have to be entered again.

- In batch mode

  Your error handler typically writes the record to an error file, reads the next input record, and continues program execution.

The DUPLICATES ARE NOT ALLOWED and VALID IF clauses are checked when the data is stored or modified. Constraints are evaluated either when the statement is executed or when the transaction is committed. The START_TRANSACTION statement EVALUATE clause allows you to specify when a constraint is to be checked. Specify VERB_TIME to detect a constraint violation while the invalid record is being processed. Specify COMMIT_TIME if the constraint depends on more than one update operation, or if immediate constraint evaluation is not critical.

The DEFINE CONSTRAINT statements are checked at the same level at which they were defined. For example, if you define constraints at the record level, and your input data involves several fields that could violate the same constraint, you will not know which field is invalid if you detect a constraint violation.

If you choose to design your input record so that no constraint can be violated by more than one field, you may increase the overhead associated with checking constraints. For example, if a relation defines constraints such that two field values in one relation cannot exist until those same field values exist in another relation, Rdb/VMS must do twice as much work to check two field constraints as to make sure one record exists in a relation before it can exist in another relation.

Alternatively you can design your program logic to check the validity of input data before you attempt the store operation.

In preprocessed and Callable RDO programs:

- To detect a violation of a constraint definition, look for the RDB$_INTEG_FAIL error code (integrity failure).
- To detect a violation of a VALID IF clause on a field definition, look for the RDB$_NOT_VALID error code.
- To detect a violation of a DUPLICATES ARE NOT ALLOWED clause of an index definition, look for the RDB$_NO_DUP error code.

10.2.5.3 Handling Fatal Errors in Preprocessed Programs In some instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error with the LIB$MATCH_COND routine or host language statements to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG, SYS$GETMSG system services, or the LIB$SIGNAL routine to output an error message.

  (The effect of the LIB$SIGNAL routine is unpredictable in some programming languages; see the following text.)
- Perform any necessary database cleanup.

- Continue program execution along the alternate path.

In any language that does define its own error handler (such as BASIC and COBOL), use of the LIB$SIGNAL routine is unpredictable. You must call the LIB$ESTABLISH, or in C, VAXC$ESTABLISH routine to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL. However, the LIB$ESTABLISH routine replaces the host language error handler in some languages. Thus, for the remainder of program execution, the host language program errors are no longer handled by the host language error handler. This means that you must explicitly handle host language errors in your condition handler. For this reason, use of the LIB$ESTABLISH routine is not recommended in host languages that have their own error handler.

In any language that does not define its own condition handler (such as FORTRAN and Pascal), you can call the LIB$SIGNAL routine to display an error message, but you must use the LIB$ESTABLISH routine (or in Pascal, the ESTABLISH function) to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

See the *VMS Run-Time Library Routines Volume* for a more complete description of the use of LIB$ESTABLISH with LIB$SIGNAL.

If you call the LIB$SIGNAL routine without establishing a condition handler, the catchall handler or the traceback handler displays the error message and terminates your program. Perform any cleanup before making the call to this routine. However, if your cleanup includes any Rdb/VMS statements, these new calls to the database will change the return status value contained in RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in that case the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling LIB$SIGNAL without any cleanup operations is not recommended.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions
- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

11 Processing Rdb/VMS Application Programs

This chapter describes how to preprocess, compile, link, and run Rdb/VMS application programs. The steps you must follow to create an executable image depend on if you want to create a shareable image and on which preprocessor you are using:

- RDBPRE
- RDML
- Callable RDO

11.1 Using the RDBPRE Preprocessor

RDBPRE is the preprocessor for BASIC, COBOL, and FORTRAN programs that contain embedded Rdb/VMS DML statements. Each DML statement is flagged by the special *&RDB&* flag, as described in Chapter 12. Following successful RDBPRE preprocessing, the resulting file is submitted automatically to the host language compiler.

Note *The RDBPRE preprocessor submits your source program to the appropriate host language compiler and creates an object file. You should never submit the output from the RDBPRE preprocessor to a host language compiler.*

When you create the source BASIC, COBOL, or FORTRAN program files, use the RDBPRE preprocessor default input file types listed in Table 11–1. You cannot use your host language default input file types. For example, do not use the COB file type for a COBOL program source file; use the preprocessor default file type RCO.

Table 11–1 shows the RDBPRE default input file types and the output file types.

Table 11–1 RDBPRE Preprocessor Default File Types

Language	RDBPRE Default Input File Type	Preprocessor Output Source File Type	List File Type
BASIC	RBA	BAS	LIS and RDBERR.LOG
COBOL	RCO	COB	LIS and RDBERR.LOG
FORTRAN	RFO	FOR	LIS and RDBERR.LOG

The LIS file is generated by the host language compiler, not by the RDBPRE preprocessor.

11.1.1 Defining Symbols to Invoke RDBPRE

You may find it easier to invoke the different RDBPRE preprocessors if you define symbols for them. For example:

```
$ RBAS :== $RDBPRE/BASIC
$ RCOB :== $RDBPRE/COBOL
$ RFOR :== $RDBPRE/FORTRAN
```

Then, use the appropriate symbol to preprocess and compile the program. In the following example, SAMPLE.RCO is an RDBPRE COBOL program that contains Rdb/VMS DML statements:

```
$ RCOB SAMPLE.RCO
```

In the preceding example, RDBPRE automatically submits the resulting SAMPLE.COB file to the COBOL compiler, which produces the SAMPLE.OBJ object module following a successful compilation.

If you do not specify the program file name, RDBPRE issues a prompt. For example:

```
$ RCOB
INPUT FILE> SAMPLE.RCO
```

In either case, the RCO file type is optional. For each language supported by RDBPRE, the RDBPRE preprocessor expects the input file type listed in Table 11–1.

You can invoke RDBPRE with the DCL RUN command, but using a defined symbol is the recommended method. An example of the RUN command is:

```
$ RUN SYS$SYSTEM:RDBPRE
INPUT FILE> SAMPLE
%RDO-F-NOPRECOMPSEL, no preprocessor selected
INPUT FILE> SAMPLE/COBOL
```

11.1.2 Using Host Language Compile Qualifiers with RDBPRE

You can add host language compile qualifiers on the command line or at the INPUT> prompt. Precede every compile qualifier with a slash (/). If you enter the compile qualifiers on the command line, you can include compile qualifiers ahead of the input file specification as well as after it. Leave at least one space between the last qualifier and the input file specification. RDBPRE does not prompt for compile qualifiers.

For example, to always preprocess BASIC programs with the /DEBUG and /LIST compile qualifiers, enter:

```
$ RBAS :== $RDBPRE/BASIC/DEBUG/LIST
$ RBAS COLLEGES.RBA
```

Note that the RDBPRE preprocessor always runs the FORTRAN compiler with the /G_FLOATING qualifier and the BASIC compiler with the /REAL_ SIZE=GFLOAT qualifier. Wrong data values will be returned by the program when all of the following conditions are true:

- A double-precision, floating-point variable is used in multiple modules.
- One module containing the variable is compiled with the G-floating size in effect (as results from being processed by the preprocessor for BASIC or FORTRAN).
- Another module containing the variable is compiled without G-floating size in effect (the default when a module is processed through a VMS host language compiler; if you do not specify G-floating, D-floating is used).
- The two modules are linked together (no error results) and pass data between each other.

No error messages are ever seen; however, wrong data answers are returned because one module passes a G-floating representation of the variable and the receiving module acts on the data as if it were D-floating. To correct this problem, always compile host language program modules specifying G-floating representation.

11.1.3 Creating RDBPRE Output Files

The RDBPRE preprocessor creates four files in your default directory when you submit a source program:

- A host language source file

  The preprocessor assigns the language default file type to the host language source file. For example, if you submitted the file PROGRAM.RCO to the RDBPRE preprocessor, it would produce the source file PROGRAM.COB.

- Two intermediate files

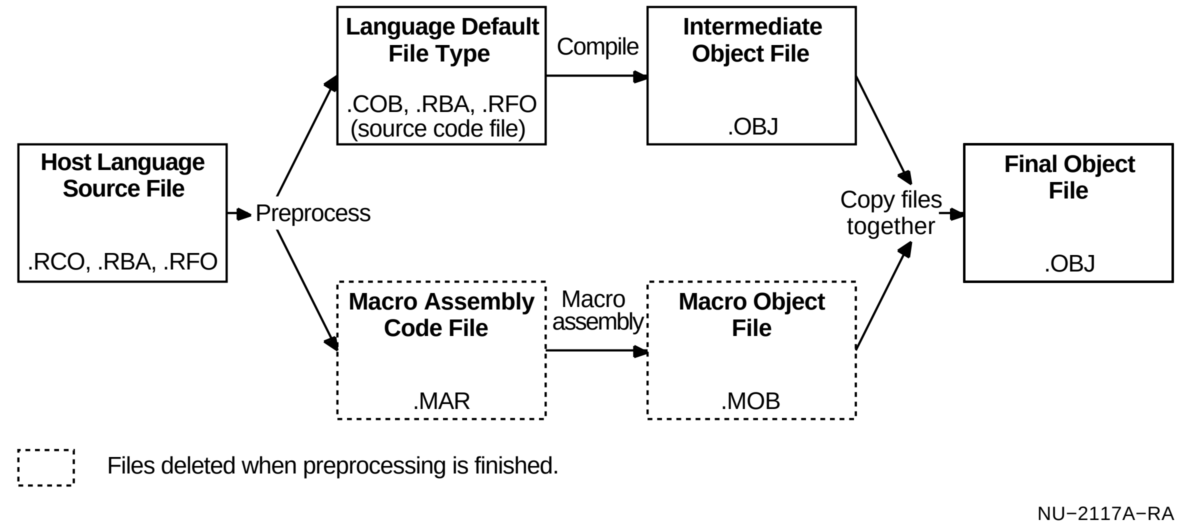
  The preprocessor creates a MAR file that contains macro assembly code that is needed by your program. The preprocessor automatically assembles this file to create an object file with the file type MOB. For example, if you submitted the file PROGRAM.RCO to the RDBPRE preprocessor, it would produce the files PROGRAM.MAR and PROGRAM.MOB. By default, both of these files are deleted from your directory when RDBPRE finishes preprocessing your source file.

- An object file from the host language source file

  This object file is the result of RDBPRE submitting your source file to the host language compiler. For example, if your source file is PROGRAM.RCO, RDBPRE would create a file called PROGRAM.OBJ.

RDBPRE creates the final object file by copying the MOB file and the intermediate object (OBJ) file into a final object file. For example, if your source file is PROGRAM.RCO, RDBPRE would append PROGRAM.MOB to PROGRAM.OBJ and the final name of this final object file would be PROGRAM.OBJ. Figure 11–1 shows this process.

Figure 11–1 Creation of an RDBPRE Object File



Object files have an OBJ file type. Using a compiler qualifier, you can instruct RDBPRE to write the object file and the source file to another directory. For example, define the following symbol:

```
$ RCOB :== $RDBPRE/COBOL
```

Now, instruct RDBPRE to write the object and source files to the PAYROLL directory:

```
$ RCOB  PROGRAMC/OBJECT=DISK3:[PAYROLL]PROGRAMC
```

RDBPRE lists the following information in the host language source file:

- Date and time the program was preprocessed
- Version of the preprocessor used
- Underlying versions of Rdb/VMS
- Command line used to preprocess the program
- Preprocessing errors, if any

Note *When changes to your program are necessary, do not edit the host language source file (BAS, COB, or FOR file type) that results from a successful RDBPRE preprocessing. As described previously, your host language source file and the intermediate files are merged in the final stages of preprocessing. If you attempt to edit the host language source file and submit it to the host language compiler, the required macro code will not be available to your program. Instead, if you need to make changes, edit the RBA, RCO, or RFO source file, then reprocess the program.*

By default, RDBPRE deletes the intermediate files (MAR and MOB) that it generates. If you wish to retain these files, enter the following command at the DCL level prompt:

```
$ DEFINE RDMS$KEEP_PREP_FILES YES
```

Do not alter the intermediate files and then attempt to produce an executable image. You should only use the intermediate files when you want to see how code has been processed by RDBPRE.

11.1.4 Displaying RDBPRE Preprocessor and Compiler Error Messages

If an error occurs during preprocessing, RDBPRE displays the error on your terminal and writes the error to the host language source file (BAS, COB, or FOR).

If an error occurs during compiling, the host language compiler displays the error on your terminal and writes the error to an error log named RDBERR.LOG. If you have used the /LIST compile qualifier, the compiler also writes the error to the list file with a LIS file extension.

11.2 Using the RDML Preprocessor

RDML statements can be embedded in C, Pascal, and VAXELN Pascal programs. These programs can be processed by the RDML preprocessor. Following a successful preprocessing, you can submit the resulting source code to the host language compiler.

You must preprocess any programs that contain RDML statements before processing them with the C or Pascal compiler. Unlike the RDBPRE preprocessor, you do not have to flag RDML statements with the &RDB& characters.

When you create the source C or Pascal program files, use the RDML preprocessor default input file types listed in Table 11–2. Table 11–2 shows the RDML default input file types and the output file types.

Table 11–2 RDML Preprocessor Default File Types

Language	RDML Default Input File Type	Preprocessor Output Source File Type	List File Type
C	RC	C	LC
Pascal	RPA	PAS	LPA

The RDML preprocessor generates a list file only if you specify the /LISTING qualifier. This is a separate list file from that produced by the host language compiler. You may find it helpful to inspect this file to see the RDML error messages that are displayed there. This can be especially helpful when you are trying to find the source of an error.

11.2.1 Defining a Symbol to Invoke RDML

Define a symbol to invoke the RDML preprocessor. For example:

```
$ RDML :== $RDML
```

or

```
$ RDML :== $RDML/PASCAL
```

or

```
$ RDML :== $RDML/C
```

If you do not use the /C or /PASCAL qualifier, you must specify the file type:

- RPA for Pascal programs
- RC for C programs

RDML will prompt for an input file if none is specified on the command line. For example:

```
$ RDML :== $RDML
$ RDML
_Source file: INVENTORY.RPA
```

An error results if you use neither a language-type qualifier nor include the default input language file type. For example:

```
$ RDML
_Source file: INVENTORY
%RDML-F-NO_LANGUAGE, No Language has been specified on the command
                     line or input file
```

11.2.2 Using RDML Qualifiers

The qualifiers for the RDML preprocessor let you control how RDML processes your files and the type of output files it produces. The format for specifying RDML qualifiers is:

```
$ RDML[qualifiers] filename
```

The RDML qualifiers are:

/C

Specifies that you want to preprocess a C source file that has embedded RDML statements.

/PASCAL

Specifies that you want to preprocess a Pascal source file that has embedded RDML statements.

/DEFAULT_TRANSACTIONS
/NODEFAULT_TRANSACTIONS

Specifies whether or not you want RDML to generate code to check the state of the database when each executable statement is executed. If you use the /DEFAULT_TRANSACTIONS qualifier (the default), RDML will generate code to attach to the database and start a read-only transaction for you when it encounters the first DML statement. Furthermore, RDML will allow you to detach from the database (issue a FINISH statement) without first closing any transaction that is attached to the database from which you are detaching (by issuing a COMMIT or ROLLBACK statement).

Use of the /DEFAULT_TRANSACTIONS qualifier can incur a significant amount of overhead; at run time, the code generated by RDML must check the state of the database and transactions as each DML statement is processed.

Use of the /NODEFAULT_TRANSACTIONS qualifier eliminates this overhead by requiring you to explicitly ready a database and start and end transactions. When you use the /NODEFAULT_TRANSACTIONS qualifier, RDML does not check the state of the database and transactions as each RDML statement is processed. In fact, if you do not close a transaction prior to issuing a FINISH statement for the database with which the transaction is associated, RDML will issue the error message: %RDB-F-OPEN_TRANS.

The default for this qualifier is /DEFAULT_TRANSACTIONS, as this maintains the behavior that RDML has always exhibited. However, Digital recommends that you always use the /NODEFAULT_TRANSACTIONS qualifier to reduce overhead and thus maximize performance.

/LISTING [=list-file-spec]
/NOLISTING

Produces a list file when you use the /LISTING qualifier. You can specify a name for the list file. If you specify the /LISTING qualifier with no file specification, the default list file specification shown in Table 11–2 is used. When you use the /NOLISTING qualifier, a list file is not produced. The /NOLISTING qualifier is the RDML default.

/OUTPUT [=output-file-spec]
/NOOUTPUT

Specifies a file for the RDML output when you use the /OUTPUT qualifier. If you specify an output file, the RDML preprocessor produces the specified output file if the source file can be preprocessed. If you use the /NOOUTPUT qualifier, no output file is produced. If you specify /OUTPUT with no file specification, the default file specification shown in Table 11–2 is used.

/LINKAGE = PROGRAM_SECTIONS

Directs RDML to communicate among separate modules using program sections when you use the /LINKAGE = PROGRAM_SECTIONS qualifier. By using this qualifier, you will be able to link your RDML modules with SQL and RDBPRE modules. The /LINKAGE = PROGRAM_SECTIONS qualifier is the RDML default.

/LINKAGE = GLOBAL_SYMBOLS

Directs RDML to generate code that communicates among separate modules using global symbols when you use the /LINKAGE = GLOBAL_SYMBOLS qualifier. Digital recommends that you use this qualifier only if you have problems linking with program sections.

/INITIALIZE_HANDLES
/NOINITIALIZE_HANDLES

Instructs RDML to generate code that automatically initializes declared database and request handles generated by RDML. This qualifier has no effect on whether or when handles are cleared in the generated code; it only controls initialization of handles in declarations. The /INITIALIZE_HANDLES qualifier is the RDML default.

Note that RDML initializes database handles (when the /INITIALIZE_HANDLES qualifier is specified or defaulted) for GLOBAL and LOCAL scope. RDML never initializes EXTERNAL scope database handles. The /INITIALIZE_HANDLES qualifier has no effect on user-declared handles. You must explicitly declare and initialize transaction and request handles that you name. When you specify the /NOINITIALIZE_HANDLES qualifier, RDML does not initialize any of the handles that RDML declares. You must initialize these handles yourself.

See Section 11.5.2 for information on when it is useful to specify the /NOINITIALIZE_HANDLES qualifier.

The RDML command defaults are /NOLISTING, /OUTPUT, /LINKAGE = PROGRAM_SECTIONS, /INITIALIZE_HANDLES, and /DEFAULT_TRANSACTIONS.

11.2.2.1 Using Host Language Compiler Qualifiers with RDML Programs

You can use any of the C and Pascal compiler qualifiers with a program that has been processed by the RDML preprocessor.

Digital recommends you always use the /G_FLOATING qualifier when you invoke the C and Pascal compilers to process source files generated by RDML. Define symbols to accomplish this. For example:

```
$ CC :== CC/G_FLOATING
$ PASCAL :== PASCAL/G_FLOATING
```

Furthermore, if the VAX C programs you generate may someday be used in other than a VMS environment, you should consider using the /STANDARD=PORTABLE qualifier with the CC command. The /STANDARD=PORTABLE qualifier causes the C compiler to generate any appropriate warning messages that flag:

- Constructs specific to C
- Deviations in your C code from conventional C constructs and rules

11.2.3 RDML Command Example

The following sequence preprocesses a program. The example assumes you have defined RDML as a foreign command, such as:

```
$ RDML :== $RDML
```

1 Create the program source file using a text editor such as VAX EDT or VAXTPU.

2 Preprocess the program with RDML:

   In Pascal:

   ```
   $ RDML/LIST SALARY.RPA
   ```

   or

   ```
   $ RDML/PASCAL/LIST SALARY
   ```

   In C:

   ```
   $ RDML/LIST INVENT.RC
   ```

   or

   ```
   $ RDML/C/LIST INVENT
   ```

3 Process RDML applications

   The source files produced by the RDML preprocessor must be compiled by the appropriate programming language compiler. Because the RDML preprocessor generates G-floating data types when representing floating-point data, Digital recommends that the /G_FLOATING qualifier be used for all modules associated with RDML preprocessed source code. This qualifier must be used when your source program contains G-floating data types. When the RDML preprocessor detects a G-floating data type, the preprocessor will issue the following reminder that the /G_FLOATING qualifier is required:

```
%RDML-I-GFLOATING, G_Floating datatypes detected in this module;
         use /G_FLOATING qualifier when compiling
```

In some cases, it is necessary that code *not* be compiled with the /G_FLOATING qualifier. This is generally only due to restrictions beyond the control of the programmer. In those rare cases, data from fields defined in the database of data type G-floating should not be accessed in the program. Mixing RDML files compiled one way with any file compiled the other way is not supported by Digital.

Compile the preprocessed program:

In Pascal:

```
$ PASCAL/G_FLOATING SALARY
```

In C:

```
$ CC/G_FLOATING INVENT
```

11.2.4 RDML Run-Time Support and Error Handling

The RDML Run-Time Library provides procedures that are used by code generated by RDML. A majority of the routines performs very low-level functions such as building argument lists, internal data transfer, and error handling. None of the present routines is of any real use to application programmers except the general purpose error handler RDML$SIGNAL_ERROR.

Digital recommends that RDML applications that require a general purpose error handler use RDML$SIGNAL_ERROR in place of other documented methods (such as RDB$SIGNAL for Rdb/VMS or RDBB$SIGNAL for Rdb/ELN). Use of RDB$SIGNAL and RDBB$SIGNAL as error handling routines will inhibit the portability of an application program on VMS systems.

The RDML$SIGNAL_ERROR routine takes a single argument, a message vector. For example:

```
RDML$SIGNAL_ERROR (RDB$MESSAGE_VECTOR);
```

The RDML$SIGNAL_ERROR routine calls the LIB$STOP routine and passes the message vector to it. If you do not specify the ON ERROR clause and an error occurs, RDML will call RDML$SIGNAL_ERROR automatically, which in turn will display all the error messages associated with the error and terminate program execution.

11.3 Using the Callable RDO Interface

Compile your source program file just as you would compile any program source file. Invoke the appropriate VAX language compiler and specify the input file specification and any compile qualifiers you want to use. For example:

```
$ COBOL/LIST/DEBUG/NOOPT TEST2
```

11.4 Creating an Executable Image: LINKING

After you have object modules for each source file, you use the LINK command to create an executable image. Your LINK command specifies all the object files you need to include in your program image.

Section 11.4.1 describes how to link Callable RDO and RDBPRE programs.

Section 11.4.2 describes how to link RDML programs.

Section 11.4.3 discusses how to link modules that were created by the RDML or RDBPRE preprocessor, the Callable RDO interface, or a combination of the three.

11.4.1 Linking Callable RDO and RDBPRE Programs

You link the object files for Callable RDO and RDBPRE programs just as you would link any program object file. Invoke the VMS Linker, and specify one or more object file specifications and any link switches you want to use.

Use a command file if you want to link in batch mode. You can use the batch job log to record any errors or warnings that occur during the link.

If your RDBPRE program uses an object library, the file name of each module in the library must be unique in the first 27 characters. Because most object file names are derived from the source file name, make sure that the conventions for naming your source files also observe this rule.

11.4.2 Linking RDML Programs

All applications generated by RDML must be linked with the RDML Run-Time Library (SYS$LIBRARY:RDMLRTL.OLB). This library contains code for various functions and procedures needed by the code generated by RDML.

To link RDML applications, use the following two lines in an options file or command file:

```
SYS$LIBRARY:RDMLRTL.OLB/LIBRARY
SYS$SHARE:VAXCRTLG.EXE/SHARE
```

For example, using an options file, type the following at the DCL prompt:

```
$ TYPE RDMLOPT.OPT
SYS$LIBRARY:RDMLRTL.OLB/LIBRARY
SYS$SHARE:VAXCRTLG.EXE/SHARE
$
```

To link an application named SALARY, type the following at the DCL prompt:

```
$ LINK SALARY,RDMLOPT/OPT
```

In a command file, enter:

```
$ TYPE MYLINK.COM
$    LINK 'P1',SYS$INPUT:/OPT
SYS$LIBRARY:RDMLRTL.OLB/LIBRARY
SYS$SHARE:VAXCRTLG.EXE/SHARE
$    EXIT
$
```

To link an application named INVENT, run the command procedure:

```
$ @MYLINK INVENT
```

11.4.3 Linking RDML Modules with RDBPRE and SQL$PRE Modules

You can link RDML, RDBPRE, and SQL$PRE modules. By specifying the /LINKAGE = PROGRAM_SECTIONS qualifier on your RDML/C or RDML/Pascal command line, you permit RDML modules to be linked with RDBPRE and SQL$PRE modules. For example, suppose you want to link an RDML/Pascal module called MY_MOD.RPA with an RDBPRE BASIC module called YOUR_MOD.RBA. You could enter the following sequence of commands to preprocess and link these two files to create an executable image:

```
$ RDML/PASCAL/LINKAGE=PROGRAM_SECTIONS MY_MOD
$ PASCAL MY_MOD.PAS
$ RDBPRE YOUR_MOD.RBA
$ LINK MY_MOD, YOUR_MOD, RDMLOPT/OPT
```

Because the /LINKAGE=PROGRAM_SECTIONS qualifier is the RDML default, it is not necessary to include it in the RDML command line. However, note that when you link RDML and RDBPRE modules you must specify the RDML options file on the LINK command line (as discussed in the section on linking RDML programs).

Note *If you have created RDML modules prior to Rdb/VMS V3.0, you must preprocess them again before you can link the object modules with RDBPRE or SQL$PRE modules. Prior to Rdb/VMS V3.0, RDML communicated between separate modules using global symbols, rather than program sections that are necessary for communication with RDBPRE and SQL$PRE modules.*

11.5 Creating a Shareable Image with RDBPRE and RDML

A shareable image file is the product of a previous linking operation. It is an image that is part of a complete program and is therefore not directly executable; that is, it is not intended to be directly executed by means of the DCL RUN command. To be used, a shareable image must be included as input in a linking operation that produces an executable image. Then, when the executable image is run, the shareable image can execute.

If you intend to have multiple releases of your applications, in which you change the application between releases to your end user, you probably want your program to be a shared image. A shared image:

- Lets you modify and enhance the contents of the shared image without requiring your end user to link his programs that attach to the shareable image each time you issue an update of your application.
- Lets the user place a single copy of the shareable image on a given node that lets all users on the node access the code contained in the shareable image. This reduces the amount of memory used on the system.

Two ways that you might want to implement shareable images in an Rdb/VMS environment are when:

- You only want the code in the shareable image to access the database.
- You want the code in the shareable image and the code in the programs that are linked against the shareable image to be able to access the database.

These two implementations are described in detail in the next two sections.

To avoid linking user programs attached to a shareable image every time you change routines in the shareable image, you should:

- Create and assemble a macro file that defines transfer vectors for your shareable image procedure entry points.

  When you link your shareable image with the /SHARE qualifier, you must place the code from this macro object file first in the shareable image. Transfer vectors provide your program with the relative address of a given routine. Example 11–1 is a simple transfer vector.
- Not rearrange or remove transfer vectors.

  If a routine is deleted from a shareable image for any reason, its transfer vector should point to a dummy routine to ensure that user programs attached to the shareable image do not fail in unforeseen ways.

- Always add new transfer vectors at the end of your transfer vector module.

  It is a good idea to allow for the addition of future transfer vectors by reserving extra space at the end of your transfer vector module. Never insert new transfer vectors between existing transfer vectors.

Example 11–1 Transfer Vector Coded for a Procedure Call

```
        .TITLE  TRANSFER_VECTORS
        .IDENT  /V1.0-001/

        .PSECT  $$$TRANSVEC,PIC,USR,CON,REL,LCL,SHR,NOEXE,RD,NOWRT,NOVEC

.TRANSFER       PROCEDURE_1           ;Begin transfer vector to PROCEDURE_1
.MASK           PROCEDURE_1           ;Store register save mask
JMP             L^PROCEDURE_1+2       ;Jump to routine, beyond the register
                                      ;save mask

.TRANSFER       PROCEDURE_2           ;Begin transfer vector to PROCEDURE_2
.MASK           PROCEDURE_2           ;Store register save mask
JMP             L^PROCEDURE_2+2       ;Jump to routine, beyond the register
                                      ;save mask


.TRANSFER       PROCEDURE_3           ;Begin transfer vector to PROCEDURE_3
.MASK           PROCEDURE_3           ;Store register save mask
JMP             L^PROCEDURE_3+2       ;Jump to routine, beyond the register
                                      ;save mask

        .END
```

For more information and examples of creating transfer vectors and linking the object file that contains them, refer to the extended VMS documentation set. The information you need is in the transfer vector section of the reference documentation for the VMS Linker utility.

To support DML statements in programs, you implicitly or explicitly specify names to identify databases, transactions, requests, declared streams, message vectors, and so forth. Names you supply explicitly are names that you supply in host language source files. Names that you supply implicitly are those that are declared by the preprocessor after your program is processed by RDBPRE or RDML. Table 11–3 shows the PSECT attributes given to the various database objects in the code generated by the RDBPRE or RDML preprocessor.

Table 11–3 PSECT Attributes Generated in RDBPRE and RDML Macro Code

Database Object	PSECT Name	PSECT Attributes Generated
Message vectors	RDB$MESSAGE_VECTOR	PIC, USR, OVR, REL, GBL, SHR, NOEXE, RD, WRT, NOVEC
LOCAL database handle User specified or default	None	No PSECT generated
GLOBAL database handle User specified or default	Same as database handle name	LONG, PIC, OVR, REL, GBL, SHR, NOEXE, RD, WRT
EXTERNAL database handle User specified or default	Same as database handle name	LONG, PIC, OVR, REL, GBL, SHR, NOEXE, RD, WRT
Default transaction handle	RDB$TRANSACTION_HANDLE	NOEXE, GBL, OVR, SHR, LONG, PIC
Preprocessor generated request handle	RDB$VARIABLES†	NOEXE, GBL, CON, SHR, LONG, PIC

†RDML does not use RDB$VARIABLES.

If you specify either a transaction or request handle, the generated macro code does not refer to it at all. You must declare it appropriately in the host language code.

You can see the PSECT attributes generated in any RDBPRE application by looking in the MAR file. Use the logical RDMS$KEEP_PREP_FILES to direct the RDBPRE preprocessor to retain the intermediate MAR files generated by the RDBPRE preprocessor. Use the DCL DEFINE command to specify that you want to retain these files:

```
$ DEFINE RDMS$KEEP_PREP_FILES YES
```

Do not alter the MAR files for any reason. If you need to make changes, make them only in your RDBPRE source file.

You can also view the PSECT attributes generated by both RDBPRE and RDML by inspecting the map file created when you link your program with the /MAP qualifier. For example, in RDBPRE:

```
$ LINK MYFILE/MAP/FULL
```

Or, in RDML:

```
$ LINK MYFILE, RDMLOPT/OPT/MAP/FULL
```

Each of these LINK commands generates a file called MYFILE.MAP that shows PSECT names and attributes.

Note that when a name is specified as global, either implicitly or explicitly, RDBPRE and RDML create overlaid PSECTs with the SHR and GBL attributes. Overlaid PSECTs with SHR and GBL attributes are appropriate for executable, but not always shareable, images. Therefore, when you link object files to create a shareable image, your LINK command should override those program sections that Rdb/VMS creates for global names. Methods of doing this are described in Section 11.5.1 and Section 11.5.2.

11.5.1 Accessing a Database from a Shareable Image Only

When you want only the code in the shareable image to access the database, you should create the code by following these steps:

1. Write the code that will be in the shareable image. This includes modules that access the database (or databases), and perhaps modules that do not.
2. Preprocess all of these modules that access the database or databases. If you are accessing a single database with a single database attach, there is no need for you to specify either transaction or database handles. If there is more than one module that accesses the database (or databases) then you must use database and transaction handles appropriately.
3. Write a transfer vector module that provides an entry point for all the routines that the shareable image provides.
4. Compile or assemble all of the modules that make up the shareable image.
5. Link the shareable image from the resulting object modules.
6. Write the code that will call the routines in the shareable image.
7. Compile or assemble the modules that make up the user's program.
8. Link these modules together against the shareable image, to produce an executable image.
9. Run the program to test it.
10. Once the shareable image has been debugged, then install it as shared (using the VMS Install utility) to make it truly shareable.

For example, suppose you write the following simple code in a file, IMAGE.RC, for use in a shareable image.

```
    DATABASE FILENAME "MF_PERSONNEL";
query()
{
FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY
            WITH E.LAST_NAME STARTING WITH "T"
            REDUCED TO E.LAST_NAME
    printf("%s, $%f\n", E.LAST_NAME, SH.SALARY_AMOUNT);
END_FOR;
}
```

To create the shareable image, you must preprocess and compile it using the following commands:

```
$ RDML IMAGE.RC
$ CC IMAGE
```

The following transfer vector module, TRANSVEC.MAR, which must be written in MACRO code, will allow an entry point into the shareable image:

```
.TITLE TRANSFER_VECTORS
.IDENT /V1.0-001/

.PSECT $$$TRANSVEC, PIC,USR,CON,REL,LCL,SHR,NOEXE,RD,NOWRT,NOVEC

.TRANSFER query
.MASK   query
.JMP     L^query+2

.END
```

Assemble it using the following command:

```
$ MACRO TRANSVEC
```

You can link these two modules, together with the necessary support routines, into a shareable image using the following DCL procedure:

```
$ LINK/SHARE=IMAGE.EXE/MAP=IMAGE.MAP/FULL TRANSVEC, IMAGE, SYS$INPUT/OPT
SYS$LIBRARY:RDMLRTL/LIBRARY
SYS$SHARE:VAXCRTLG/SHARE
PSECT_ATTR=RDB$DBHANDLE, NOSHR, LCL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR, LCL
PSECT_ATTR=RDB$MESSAGE_VECTOR, NOSHR, LCL
```

In addition, if any global database or transaction handles are used in the subprograms in the shared image, you will need another line:

```
PSECT_ATTR=<db_handle_name>,NOSHR,LCL
```

or

```
PSECT_ATTR=<transaction_handle_name>,NOSHR,LCL
```

Include a line using these formats for each database handle or transaction handle, where db_handle_name is the name of the global database handle and transaction_handle_name is the name of the transaction handle.

Note the PSECT_ATTR options. These are necessary for the message vector and for every database and transaction handle used. Here, only the default database and transaction handles are used. The NOSHR option for these handles and the message vector must be specified so that these variables are specific to each process, and not shared among processes. If they are shared among processes, then the processes will interfere with each other in their use of handles and error codes. The LCL attribute ensures that the PSECTs can only be used in the shareable image. (This attribute should only be used when you do not want to access the database from both the shareable image and the main images.)

Note that the transfer vector PSECT *must* be the first PSECT in the shareable image because its location must not change in subsequent versions of the application. The previous example relies on the name and attributes of the PSECTs for placing the PSECT in the image; it could also use the CLUSTER and COLLECT options in the LINKER options file.

When you create a shareable image, all Rdb/VMS shared information, such as transaction handles, database handles, and message vectors, should be set to NOSHR. In addition, host language compilers may also create overlaid program sections for global names. You may need to specify additional PSECT_ATTR lines to set to NOSHR other parameters you define as global or external in your program. For example, you may need to specify PSECT_ATTR lines for common areas. (Common areas are created by FORTRAN COMMON statements, BASIC MAP and COMMON declarations, COBOL EXTERNAL working storage variables, and PASCAL variables with the [COMMON] attribute.

To check that the attributes are set correctly, you can look at the linker map of the image. The following example links MYPROG and produces a .MAP file:

```
$ LINK/MAP=TEST.MAP/NOEXEC MYPROG
```

Then, search the .MAP file for elements that have the program section attributes of GLB and SHR, as shown in the following example (note that there are two spaces between GBL and SHR):

```
 $ search test.map "GBL,  SHR"

RDB$DBHANDLE                    00000608 0000061C 00000015 (         21.)
2
   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT,NOVEC
RDB$MESSAGE_VECTOR              00000620 0000066F 00000050 (         80.)
2
   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT,NOVEC
RDB$TRANSACTION_HANDLE          00000670 00000677 00000008 (          8.)
2
   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT,NOVEC
```

If Rdb/VMS shared information and common areas used by your application have the SHR attribute, change them to NOSHR using the PSECT_ATTR options for the LINK command.

Once the shareable image has been created, a simple program can be written to call it. For example:

```
extern void query();

main()
{
    query();
}
```

Suppose you name the file that contains this code MAIN.C. Compile the file using the command:

```
$ CC MAIN
```

Link it against the shareable image with this command (assuming that IMAGE.EXE is in your default directory):

```
$ LINK/DEB/MAP=MAIN.MAP/FULL MAIN.OBJ, SYS$INPUT/OPT
IMAGE/SHARE
```

Note that the program has no need to look at any handles or message vectors. They are hidden from it by the shareable image interface.

In order to run the program, for debugging purposes, you must define a logical name, IMAGE, to point to the shareable image you built. Otherwise the image activator will search SYS$SHARE for an IMAGE.EXE file:

```
$ DEFINE IMAGE SYS$DISK:[]IMAGE
```

Now you can run the program with the following command:

```
$ RUN MAIN
```

Once you have debugged your shareable image, you can place the working version of IMAGE.EXE in SYS$SHARE (install it as shared using the VMS Install utility), and can deassign your IMAGE logical name. Your users can then link against it as follows:

```
$ LINK/DEB/MAP=MAIN.MAP/FULL MAIN.OBJ, SYS$INPUT/OPT
SYS$SHARE:IMAGE/SHARE
```

11.5.2 Accessing a Database from a Shareable Image and an Application Program

If you want both the shareable image and the program that is linked against the shareable image to access the database, you must take a slightly different approach from that mentioned in Section 11.5.1. RDBPRE and RDML provide the /NOINITIALIZE_HANDLES qualifier that lets you alter the method used in Section 11.5.1 so that the shareable image and the program that calls it can access the database.

Note *In RDBPRE when you use the /NOINITIALIZE_HANDLES qualifier, any handle you specify in your application program must also be specified in the shareable image.*

The steps involved in accessing a database from a shareable image and an application program are:

1. Write the code that will be in the shareable image. This includes modules that access the database (or databases), and perhaps modules that do not.
2. Preprocess all of these modules that access the database (or databases). If you are accessing a single database with a single database attach, there is no need for you to specify either transaction or database handles. If there is more than one module that accesses the database (or databases), then you must use database and transaction handles appropriately.
3. Write a transfer vector module that provides an entry point for all the routines that the shareable image provides.
4. Compile or assemble all of the modules that make up the shareable image.
5. Link the shareable image from the resulting object modules.
6. Write the code that will call the routines in the shareable image. This consists of a number of modules, some that access the database directly, and some that do not.
7. Preprocess the modules that access the database.
8. Compile or assemble the modules that make up the user's program.
9. Link these modules together against the shareable image, to produce an executable image.
10. Run the program to test it.
11. Once the shareable image has been debugged, then install it as shared (using the VMS Install utility) to make it truly shareable.

Assume you are going to use the same code in the shareable image as that developed in Section 11.5.1 for the logical name IMAGE. For use with a program module that also accesses the database, you need to preprocess and compile the code that you plan to use in the shareable image, just as was done in Section 11.5.1. You can also use the same transfer vector. However, in this case, when you link the code for the shareable image do *not* specify LCL for the PSECT attributes because you want the shareable image and the program that uses the shareable image to share the PSECTs. See the following RDML example.

```
$ LINK/SHARE=IMAGE.EXE/MAP=IMAGE.MAP/FULL TRANSVEC, IMAGE, SYS$INPUT/OPT
SYS$LIBRARY:RDMLRTL/LIBRARY
SYS$SHARE:VAXCRTLG/SHARE
PSECT_ATTR=RDB$DBHANDLE, NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR
PSECT_ATTR=RDB$MESSAGE_VECTOR, NOSHR
```

In addition, if any global database or transaction handles are used in the subprograms in the shared image, you will need another line:

```
PSECT_ATTR=<db_handle_name>,NOSHR
```

or

```
PSECT_ATTR=<transaction_handle_name>,NOSHR
```

Include a line using these formats for each database handle or transaction handle, where db_handle_name is the name of the global database handle and transaction_handle_name is the name of the transaction handle.

Now that you have created a shareable image, you need to write the program that will call the shareable image. In RDML programs, you can begin the process of creating a shareable image that will let both the shareable image and the calling RDML program access a database by adding a small amount of code to the MAIN.RC module (discussed in Section 11.5.1).

```
DATABASE FILENAME "PERSONNEL";

extern void query();

main()
{
    /* Call code in the shareable image to do a query. */

    query();

    /* Now do a query in this module. */

    FOR FIRST 5 E IN EMPLOYEES CROSS D IN DEGREES
                   WITH D.DEGREE STARTING WITH "B"
            printf("%s, %s : %s\n", E.LAST_NAME, E.FIRST_NAME, D.DEGREE);
    END_FOR;
}
```

The program is still using the default database and transaction handles, only. Preprocess it with the following command:

```
$ RDML/NOINITIALIZE_HANDLES MAIN.RC
```

Note the /NOINITIALIZE_HANDLES qualifier. It stops RDML from initializing handles to zero, which is normally done when you preprocess code with RDML. Because the program will be using the default handles that were defined (as PSECTs) in the shareable image, an attempt to initialize them to zero would result in a LINKER error because the PSECT does not exist in the image you are going to build. If you fail to use the /NOINITIALIZE_

HANDLES qualifier, you will receive an error similar to the following when you attempt to link the program with the shareable image:

```
%LINK-E-OUTSIMG, attempted store location %X00000000 is outside
image binary (%X00000000 to %X00000000)
in psect DBH module RDB$DMTST FILE DISK4:[SHAREDIR.TEMP]TEST.OBJ;7
-LINK-E-NOIMGFIL, image file not created
```

Compile MAIN.RC with this command:

```
$ CC MAIN
```

Link it with this command:

```
$ LINK/DEB/MAP=MAIN.MAP/FULL MAIN.OBJ, SYS$INPUT/OPT
SYS$LIBRARY:RDMLRTL/LIBRARY
IMAGE/SHARE
PSECT_ATTR=RDB$DBHANDLE, NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR
PSECT_ATTR=RDB$MESSAGE_VECTOR, NOSHR
```

In addition, if any global database or transaction handles are used in the subprograms in the shared image, you will need another line:

```
PSECT_ATTR=<db_handle_name>,NOSHR
```

or

```
PSECT_ATTR=<transaction_handle_name>,NOSHR
```

Include a line using these formats for each database handle or transaction handle, where db_handle_name is the name of the global database handle and transaction_handle_name is the name of the transaction handle.

Note the NOSHR attribute specified in the PSECTs. This is done so that the PSECT attributes match those of the shareable image. Also, it is not necessary to link against SYS$LIBRARY:VAXCRTLG.EXE because the shareable image is already linked against that. Do not change the GBL attributes of the PSECTs to LCL in this case, because you want the PSECTs to be shared across images.

Now you can run the new MAIN program (remembering the IMAGE logical name again) with the following command:

```
$ RUN MAIN
```

If you use your own transaction or database handles, then you must declare them to produce a PSECT of matching name and attributes, as RDB$TRANSACTION_HANDLE was declared in the previous example. You can look at the declaration of RDB$TRANSACTION_HANDLE in the RDMLVAXC.H or RDMLVPAS.PAS files in SYS$LIBRARY for how to do this in C or Pascal.

11.6 Running the Program

Execute the EXE program created by either of the preprocessors or Callable RDO with the DCL RUN command. For example:

```
$ RUN JOBHIST.EXE
```

11.7 Debugging with the VMS Debugger

The VMS Debugger provides a convenient way for you to monitor the execution of your program at run time. With the debugger you can:

- Step through the program one statement at a time
- Examine and modify statements and data values
- Stop program execution at specified points
- Display messages at specified points in the program

Use the /NOOPTIMIZE compile qualifier when you use the /DEBUG compile qualifier. The default for many VMS compilers, /OPTIMIZE, causes the compiler to optimize the compiled program to generate more efficient code. Thus, unless you specify the /NOOPTIMIZE qualifier, the code you try to debug may be different from your original code.

To use the VMS Debugger with RDBPRE programs you must specify the /DEBUG qualifier:

- On the preprocess command line

  For example:

  ```
  $ RDBPRE/BASIC PROGRAM.RBA/DEBUG
  ```

- On the LINK command line

  For example:

  ```
  $ LINK PROGRAM/DEBUG
  ```

To use the VMS Debugger with RDML programs you must specify the /DEBUG qualifier:

- On the compiler command line

  For example:

  ```
  $ RDML/PASCAL PROGRAM.RPA
  $ Pascal PROGRAM/DEBUG
  ```

- On the LINK command line

  For example:

  ```
  $ LINK PROGRAM,RDMLOPT/OPT/DEBUG
  ```

For detailed information about using the debugger, refer to the *VMS Debugger Manual* or to the chapter in your particular language user's guide that describes how to use the debugger.

12 Using the RDBPRE Program Environment

This chapter describes how to develop applications using RDBPRE preprocessed programs. The chapter presents the following topics:

- Differences in syntax between RDO and RDBPRE
- Using data manipulation statements in the program environment
- Copying CDD/Plus definitions to declare host language variables

Most of the information you need to develop an RDBPRE program is contained in Chapter 9 and Chapter 13 through Chapter 15. This chapter serves to provide you with information that is specific to RDBPRE and applies to all the RDBPRE programming languages.

12.1 RDBPRE Program Development

To ensure effective program development you should:

- Develop your queries in RDO

  You need to know which databases, relations, and fields your program accesses. Special characteristics of the relations, views, and field definitions in the database will determine the most efficient form for a query. A discussion about developing query prototypes in RDO is provided in Chapter 7.

- Determine host language variables

  Your program usually needs to declare host language variables that pass values to and accept values from the database. You need to be aware of the existing data types, data restrictions, and input constraints that are part of the design of the databases you access. A discussion of determining host language variables is provided in Chapter 7. Data type conversions for the RDBPRE preprocessor languages are provided in Chapter 8.

- Convert your query to the program environment

  In a typical application, the database is the source of records for reports and calculations and the target for updates. The host language provides logic for operations such as loops, conditional processing, numeric manipulation, and input/output operations. The next section discusses how to convert a prototype to a host language program.

12.1.1 Differences in RDO and RDBPRE Data Manipulation Language Syntax

The RDBPRE data manipulation statements are a subset of the RDO statements. With the RDBPRE statements you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. However, no RDBPRE statements exist to perform data definition tasks. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of the Rdb/VMS data manipulation statements.

The syntax you use for preprocessed Rdb/VMS statements is not identical to the statement syntax you use in RDO. When you convert RDO syntax to RDBPRE syntax, be aware that you:

- Use the GET statement instead of the PRINT statement
- Can nest FETCH and GET operations within a host language loop
- Can use the ON ERROR and AT END clauses to identify Rdb/VMS errors

Examples of how to use these statements are contained in Chapter 13 through Chapter 15.

12.1.2 Using the &RDB& Statement Flag

The preprocessor must be able to distinguish between Rdb/VMS data manipulation statements and host language statements. The Rdb/VMS statement flag signals Rdb/VMS data manipulation statements. A statement flag consists of the letters RDB between two ampersands (&RDB&). The flag must be the *first* nonblank character on the program line. Leading and trailing spaces and tabs are optional unless your host language requires otherwise.

All lines in a multiple-line Rdb/VMS statement should be flagged. The preprocessor will accept the host language continuation character to continue Rdb/VMS statements to new lines, but use of host language continuation characters instead of Rdb/VMS statement flags can create confusing code. When every line of a multiple-line Rdb/VMS statement is flagged, the Rdb/VMS statement is clearly distinguishable from the surrounding host language statements.

Your RDBPRE program cannot mix Rdb/VMS statements and host language statements on the same program line. Thus you cannot use an Rdb/VMS statement flag on any program line that also includes a host language statement. The *only* host language element that can appear on the same line as an Rdb/VMS statement is a host language variable. The proper position of the host language variable is governed by Rdb/VMS syntax. You will receive one or more preprocessor errors if you place a BASIC line number, COBOL label, or FORTRAN statement label on the same line as the &RDB& statement flag. For example, if you place a BASIC line number between the flag and the DATABASE statement, you may receive errors such as the following:

```
&RDB&  20 DATABASE GLOBAL pers = FILENAME "PERSONNEL" DBKEY SCOPE IS FINISH
*         ^
* *** ERROR, one of the following symbols was expected:
;           FOR GET ERASE FETCH PSECT READY STORE COMMIT FINISH INVOKE
            MODIFY PREPARE DATABASE ROLLBACK END_STREAM START_STREAM
;           START_TRANSACTION END_SEGMENTED_STRING START_SEGMENTED_STRING
            CREATE_SEGMENTED_STRING _|_
```

If you omit the &RDB& statement flag from any Rdb/VMS statement that ends a block of statements, such as END_FOR, END_GET, END_STORE, or END_MODIFY, the preprocessor issues the error message: "**** ERROR, statement is syntactically incomplete". It is a good idea to check the placement of &RDB& statement flags first when locating preprocessing errors.

The following example shows the format and location of the statement flag within an Rdb/VMS statement. The statements preceded by the &RDB& statement flag can be embedded in your host language as they appear here. The statements that appear in lowercase must be converted to your host language. This will be true for all the examples in the BASIC, COBOL, and FORTRAN language chapters.

```
!print supervisor ID, name, department code

        &RDB&  FOR C IN CURRENT_JOB CROSS SC IN CURRENT_JOB
        &RDB&    WITH C.EMPLOYEE_ID = Id
        &RDB&    AND C.SUPERVISOR_ID = SC.EMPLOYEE_ID
        &RDB&      ON ERROR
                     error handling statements
        &RDB&      END_ERROR
        &RDB&    GET
        &RDB&        ON ERROR
                       error handling statements
        &RDB&        END_ERROR
        &RDB&      Super_id = C.SUPERVISOR_ID;
        &RDB&      Last_name = SC.LAST_NAME;
        &RDB&      Super_dept = SC.DEPARTMENT_CODE
        &RDB&    END_GET

                 if (Super_id <> Id)
                   then
                     print  Super_dept, Super_id, Last_name
        &RDB&    END_FOR
```

12.1.3 Copying Data Dictionary Definitions to Declare Host Language Variables

A convenient way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus. You can copy field and relation definitions, which include all the fields within the relation. However, you must be careful to copy only those relation and field definitions whose data types are supported by your host language.

You can use the data dictionary to copy the data definitions of database fields and relations into your program. Using the data dictionary:

- Simplifies the task of program data definition
- Automatically defines all the fields within a relation
- Ensures that all programs that use the database define the same field consistently
- Provides your program with the correct host language data type, unless the database data type is unsupported by your host language

BASIC, COBOL, and FORTRAN let you copy data dictionary definitions into your program. However, copying from the data dictionary is not a completely automatic process for the programmer. Be careful to avoid the following:

- Naming conflicts

  You must ensure that relation and field names copied into your program do not conflict with the BASIC, COBOL, or FORTRAN naming rules and are not BASIC, COBOL, or FORTRAN reserved words. If there are any naming conflicts, you must change the name in the appropriate database definition before copying the data dictionary definition. For information about changing database attribute definitions, see the *VAX Rdb/VMS RDO and RMU Reference Manual*.

- Field names that are not unique

  Field definitions copied from the data dictionary are likely to contain field names that are not unique. Be sure to qualify any of these field names by the relation name that contains it. If you do not qualify field names that are not unique, you will get compile-time errors indicating ambiguous references. You can use the appropriate compiler qualifier to print out the copied definition or definitions in your list (LIS) file and then you can check for ambiguous field names.

- Data type conflicts

  The BASIC, COBOL, and FORTRAN processors translate copied data dictionary data types into equivalent host language data types where possible. The BASIC, COBOL, and FORTRAN processors, however, do not perform the data type conversions that the Rdb/VMS preprocessor performs. If an Rdb/VMS data type is flagged with a dagger in Table 8–4, Table 8–6, or Table 8–7, you should check the conversion performed by the data dictionary and make sure that the data type is appropriate for your application.

As stated previously, you can copy field and relation definitions (and the definitions for the fields contained within the relation) from the data dictionary. Relation definitions are stored in the data dictionary as objects under the directory RDB$RELATIONS. Field definitions are stored in the data dictionary as objects under the directory RDB$FIELDS. To copy a relation into your program, specify the location of your database dictionary, then specify the database and the dictionary path name of the relation that you want to copy.

For example, to copy the relation EMPLOYEES from an MF_PERSONNEL database, specify:

```
DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES
```

The data dictionary is organized as a hierarchy of directories and objects. You can use the Common Dictionary Operator utility (CDO) to display on your terminal all the data dictionary objects for a particular database. By observing the display, you can identify the data dictionary path name you need to include in the copy statement. For additional information about the data dictionary structure, see the *VAX Common Data Dictionary Utilities Reference Manual*.

To list field definitions on your terminal, you must first use the CDO ENTER command to assign CDO directory names to relation definitions within a CDD$DATABASE definition. Then enter the CDO SHOW ALL/FULL command to see the field definitions for all the relations that you have assigned CDO directory names with the ENTER command.

For example, if you want to see the field definitions for the EMPLOYEES relation in the MF_PERSONNEL database, enter the following commands:

```
$ DICTIONARY OPERATOR
CDO> ENTER RECORD EMPLOYEES FROM DATABASE MF_PERSONNEL
CDO> SHOW ALL/FULL
Definition of record EMPLOYEES
|   Contains field          EMPLOYEE_ID
|   |   Based on                ID_NUMBER
|   |   |   Description             ’ Generic employee ID ’
|   |   |   Datatype                text size is 5 characters
|   Contains field          LAST_NAME
|   |   Description             ’ Generic last name ’
|   |   Datatype                text size is 14 characters
|   Contains field          FIRST_NAME
|   |   Description             ’ Generic first name ’
|   |   Datatype                text size is 10 characters
|   Contains field          MIDDLE_INITIAL
|   |   Description             ’ Generic middle initial ’
|   |   Datatype                text size is 1 characters
|   |   Missing_value           " "
|   |   DTR Edit_string         X.
|   Contains field          ADDRESS_DATA_1
|   |   Description             ’ Street name ’
|   |   Datatype                text size is 25 characters
|   |   Missing_value           "                         "
|   Contains field          ADDRESS_DATA_2
|   |   Description             ’ Mail stops, suite addresses, street
|   |                             numbers, and so forth.’
|   |   Datatype                text size is 25 characters
|   |   Missing_value           "                         "
|   Contains field          CITY
|   |   Description             ’ City name ’
|   |   Datatype                text size is 20 characters
|   |   Missing_value           "                    "
|   Contains field          STATE
|   |   Description             ’ State abbreviation (or DISTRICT) ’
|   |   Datatype                text size is 2 characters
|   |   Missing_value           "  "
|   Contains field          POSTAL_CODE
|   |   Description             ’ Postal code (in US = ZIP)’
|   |   Datatype                text size is 5 characters
|   |   Missing_value           "     "
|   Contains field          SEX
|   |   Description             ’ M, F ’
|   |   Datatype                text size is 1 characters
|   |   Missing_value           "?"
|   |   Valid if                (((SEX EQ "M") OR (SEX EQ "F")) OR
|   |                             (SEX MISSING))
|   Contains field          BIRTHDAY
|   |   Based on                STANDARD_DATE
|   |   |   Description             ’ Generic date field ’
|   |   |   Datatype                date
|   |   |   Missing_value           17-NOV-1858 00:00:00.00
|   |   |   DTR Edit_string         DD-MMM-YYYY
|   Contains field          STATUS_CODE
|   |   Description             ' A number '
|   |   Datatype                text size is 1 characters
|   |   Missing_value           "N"
|   |   Valid if                ((((STATUS_CODE EQ "0") OR
                                   (STATUS_CODE EQ "1")) OR
                                   (STATUS_CODE EQ "2")) OR
                                   (STATUS_CODE MISSING ))

Definition of database  MF_PERSONNEL
|   database uses RDB database MF_PERSONNEL
|   database in file MF_PERSONNEL
|   |   fully qualified file DISK:[MYDATABASE]MF_PERSONNEL.RDB;
CDO>
```

12.1.3.1 The INCLUDE Directive in BASIC After identifying the data dictionary path name of the relation you want to copy, use the BASIC %INCLUDE %FROM %CDD directive to copy the definitions into your program. To copy multiple definitions, repeat the %INCLUDE directive as needed.

The format of the INCLUDE directive is:

```
%INCLUDE %FROM %CDD 'dictionary-path-name'
```

The argument for the INCLUDE directive is dictionary-path-name.

The full or relative data dictionary path name that identifies the location in the data dictionary of the object definition you want to copy. Enclose the path name in single quotation marks. You can use a logical name for dictionary-path-name.

The following program segment shows you how to copy a data dictionary definition into a BASIC program. Note that BASIC translates the data dictionary definition into a record structure and subordinate fields. The BASIC RECORD statement is only a structure and does not assign memory. To allocate memory to the fields of the record, you must declare a variable of that record type. Line 10 of the following example declares the record and line 20 defines the record structure by copying the definition from the data dictionary:

```
   !
   !Declare the record.
   !
10 DECLARE EMPLOYEES EMP_REC
   !
   !Copy VAX CDD/Plus data definition.
   !
20 %INCLUDE %FROM %CDD 'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'
```

You can obtain a processed program list (LIS) file that includes the translated definition using the /LIST and /SHOW = CDD_DEFINITIONS qualifiers in the process command line. Include these qualifiers when you invoke the Rdb/VMS preprocessor, RDBPRE; or define the preprocess command to include these and other qualifiers, as shown in the next example.

```
$ RDBPRE/BASIC PROG2/LIST/SHOW=CDD_DEFINITIONS
```

The following list segment from the LIS file shows the BASIC translated text of the EMPLOYEES relation definition (with control characters removed to improve readability). Note the data type conversion for the DATE data type of the field BIRTHDAY. See Chapter 13 for information on using DATE data types in RDBPRE BASIC programs.

```
15 DECLARE EMPLOYEE EMP_REC
20 %INCLUDE %FROM %CDD 'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'
        C1              RECORD  EMPLOYEES                 ! UNSPECIFIED
        C1                STRING  EMPLOYEE_ID  = 5        ! TEXT
        C1                !   Generic employee ID
        C1                STRING  LAST_NAME  = 14         ! TEXT
        C1                !   Generic last name
        C1                STRING  FIRST_NAME  = 10        ! TEXT
        C1                !   Generic first name
        C1                STRING  MIDDLE_INITIAL  = 1     ! TEXT
        C1                !   Generic middle initial
        C1                STRING  ADDRESS_DATA_1  = 25    ! TEXT
        C1                !   Street name
        C1                STRING  ADDRESS_DATA_2  = 25    ! TEXT
        C1                !   Mail stops, suite addresses, street
C1                        !   numbers, and so forth.
        C1                STRING  CITY  = 20              ! TEXT
        C1                !   City name
        C1                STRING  STATE  = 2              ! TEXT
        C1                !   State abbreviation (or DISTRICT)
        C1                STRING  POSTAL_CODE  = 5        ! TEXT
        C1                !   Postal code (in US = ZIP)
        C1                STRING  SEX  = 1                ! TEXT
        C1                !   M, F
        C1                GROUP   BIRTHDAY                ! DATE
        C1                !   Generic date field
        C1                  STRING  STRING_VALUE  = 8
        C1                END GROUP
        C1                STRING  STATUS_CODE  = 1        ! TEXT
        C1                !   A number
        C1              END RECORD
...................1
%BASIC-I-CDDSUBGRO, 1:      data type in CDD not supported, substituted group
                            for: EMPLOYEES::BIRTHDAY.
```

12.1.3.2 The COPY FROM DICTIONARY Statement in COBOL After identifying the data dictionary path name of the relation you want to copy, use the COBOL COPY FROM DICTIONARY statement to copy the definitions into the WORKING-STORAGE section of your program. Repeat the COPY FROM DICTIONARY statement as needed.

You can replace a field name in the definition by using the REPLACING clause in the COPY FROM DICTIONARY statement. When you copy a field definition from the data dictionary, the field appears in your program as a level 01 declaration. If the data dictionary field name is not unique, replace it with a unique level 01 field name.

The format of the COPY FROM DICTIONARY statement is:

```
COPY "dictionary-path-name" FROM DICTIONARY
      [REPLACING dictionary-field-name BY new-field-name, ...].
```

The arguments from the COPY FROM DICTIONARY statement are:

- dictionary-path-name

  The full or relative data dictionary path name that identifies the location in the data dictionary of the object definition you want to copy. Enclose the path name in pairs of single or double quotation marks. You can use a logical name for dictionary-path-name.

- dictionary-field-name

  The CDD/Plus field name you want to replace with a new name.

- new-field-name

  The field name you want to substitute for the CDD/Plus field name. Use the REPLACING clause to provide unique field names when you do not want to qualify the field name or when the field is a level 01 declaration.

  The following program segment copies the data dictionary definitions for the EMPLOYEES relation within the MF_PERSONNEL database. Note that COBOL translates the data dictionary relation definition into a level 01 record with level 02 fields.

```
WORKING-STORAGE SECTION.
*
*Copy data dictionary relation definitions.
*
 COPY "DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES"
             FROM DICTIONARY.
```

  You can obtain a processed program list (LIS) file that includes the translated definition using the /LIST and /COPY_LIST qualifiers in the process command line. Include these qualifiers when you invoke the Rdb/VMS preprocessor, RDBPRE; or define the preprocess command to include these and other qualifiers. For example:

```
$ RDBPRE :== $RDBPRE
$ RDBPRE/COBOL PROG2/LIST/COPY_LIST
```

  The following list segment from the LIS file shows the COBOL translated text of the EMPLOYEES relation definition. Note the data type conversion for the DATE data type of the field BIRTHDAY. See Chapter 14 for information on using DATE data types in RDBPRE COBOL programs.

```
  324         DATA DIVISION.
  325         WORKING-STORAGE SECTION.
  326        * copy VAX CDD/Plus definitions in COBOL
  327
  328         COPY "DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES"
  329               FROM DICTIONARY.
  330L        *
  331L        * DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES
  332L        *
  333L        01  EMPLOYEES.
  334L        *  Generic employee ID
  335L             02  EMPLOYEE_ID         PIC X(5).
  336L        *  Generic last name
  337L             02  LAST_NAME           PIC X(14).
  338L        *  Generic first name
  339L             02  FIRST_NAME          PIC X(10).
  340L        *  Generic middle initial
  341L             02  MIDDLE_INITIAL      PIC X.
  342L        *  Street name
  343L             02  ADDRESS_DATA_1      PIC X(25).
  344L        *  Mail stops, suite addresses, street numbers, and so forth.
  345L             02  ADDRESS_DATA_2      PIC X(25).
  346L        *  City name
  347L             02  CITY                PIC X(20).
  348L        *  State abbreviation (or DISTRICT)
  349L             02  STATE               PIC X(2).
  350L        *  Postal code (in US = ZIP)
  351L             02  POSTAL_CODE         PIC X(5).
  352L        *  M, F
  353L             02  SEX                 PIC X.
  354L        *  Generic date field
  355L             02  BIRTHDAY            PIC S9(11)V9(7) COMP.
                     1
%COBOL-W-ERROR  405, (1) Absolute date and time datatype represented in
                         one second units - PIC S9(11)V9(7) COMP assumed
  356L        *  A number
  357L             02  STATUS_CODE         PIC X.
  358
```

12.1.3.3 The DICTIONARY Statement in FORTRAN

FORTRAN lets you copy database data definitions from the data dictionary into your FORTRAN program. After identifying the data dictionary path name of the relation you want to copy, use the FORTRAN DICTIONARY statement to copy the definitions into your program. Repeat the DICTIONARY statement as needed.

The format of the DICTIONARY statement is:

```
DICTIONARY 'dictionary-path-name'
```

The argument for the DICTIONARY statement is dictionary-path-name.

Dictionary-path-name refers to the full or relative data dictionary path name that identifies the location in the data dictionary of the object definition you want to copy. Enclose the path name in pairs of single quotation marks. You can use a logical name for dictionary-path-name.

The following program segment shows you how to copy a data dictionary definition into a FORTRAN program. Note that FORTRAN translates the data dictionary definition into a record structure and subordinate fields. To allocate memory to the fields of the record, you must declare the record with the RECORD statement.

```
C      Declare record

       RECORD /EMPLOYEES/EM

C      Copy VAX CDD/Plus definition

       DICTIONARY
      1  'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'
```

You can obtain a processed program list (LIS) file that includes the translated definition using the /LIST and /SHOW=DICTIONARY qualifiers in the process command line. Include these qualifiers when you invoke the Rdb/VMS preprocessor, RDBPRE; or define the RDBPRE command to include these and other qualifiers (see Chapter 11). For example:

```
$ RDBPRE/FORTRAN PROG2/LIST/SHOW=DICTIONARY
```

The following list segment from the LIS file shows the FORTRAN translated text of the EMPLOYEES relation definition. Note the data type conversion for the DATE data type of the field BIRTHDAY. See Chapter 15 for information on using DATE data types in RDBPRE FORTRAN programs.

```
0003    C     Copy data dictionary definition
0004
0005          DICTIONARY 'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'
0006  1     ! CDD Path Name "DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPL
0007  1     ! OYEES"
0008  1     STRUCTURE /EMPLOYEES/
0009  1         !  Generic employee ID
0010  1         CHARACTER*5 EMPLOYEE_ID
0011  1         !  Generic last name
0012  1         CHARACTER*14 LAST_NAME
0013  1         !  Generic first name
0014  1         CHARACTER*10 FIRST_NAME
0015  1         !  Generic middle initial
0016  1         CHARACTER*1 MIDDLE_INITIAL
0017  1         !  Street name
0018  1         CHARACTER*25 ADDRESS_DATA_1
0019  1         !  Mail stops, suite addresses, street numbers, and so forth.
0020  1         CHARACTER*25 ADDRESS_DATA_2
0021  1         !  City name
0022  1         CHARACTER*20 CITY
0023  1         !  State abbreviation (or DISTRICT)
0024  1         CHARACTER*2 STATE
0025  1         !  Postal code (in US = ZIP)
0026  1         CHARACTER*5 POSTAL_CODE
0027  1         !  M, F
0028  1         CHARACTER*1 SEX
0029  1         !  Generic date field
0030  1         STRUCTURE BIRTHDAY
%FORT-I-UNSUPPTYPE, CDD description specifies an unsupported datatype.
        in module F_SAMPLE$MAIN at line 30
```

```
0031  1             LOGICAL*1 %FILL (8)
0032  1         END STRUCTURE
0033  1         !  A number
0034  1         CHARACTER*1 STATUS_CODE
0035  1     END STRUCTURE
0036
0037    C    Declare record
0038
0039           RECORD / EMPLOYEES/EM
0040
```

13

Using the BASIC Program Environment

This chapter describes how to access an Rdb/VMS database using BASIC and the Rdb/VMS preprocessor, RDBPRE. This chapter presents the following main topics:

- Using Rdb/VMS data manipulation statements
- Using Rdb/VMS data definition statements
- Error handling in RDBPRE BASIC

Most examples in this chapter are available on line. The Rdb/VMS installation procedure writes the sample programs to SYS$COMMON:[SYSHLP.EXAMPLES.RDBVMS]. The file names for these programs are: B_SAMPLE.RBA, B_CALL_OTHER.RBA, B_CALLABLE_ERROR_HANDLER.BAS and B_ERROR_HANDLER.BAS. The sample program B_SAMPLE.RBA contains most of the procedures referred to in this chapter.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

Note *Before reading this chapter, you should be familiar with the information contained in Chapter 9. The main purpose of this chapter is to provide information and examples specific to VAX BASIC.*

13.1 The RDBPRE BASIC Preprocessor Interface

When you use the RDBPRE BASIC preprocessor interface, you simply include Rdb/VMS data manipulation statements directly in your program wherever you need them. You must use the special statement flag (&RDB&) with each Rdb/VMS data manipulation statement you include in your BASIC program. When you preprocess the source program, the preprocessor converts the Rdb/VMS data manipulation statements to a series of BASIC calls to Rdb/VMS. At run time, Rdb/VMS executes the calls and returns any retrieved data to the program.

The first line of an RDBPRE BASIC program cannot be a blank line or a line that continues to the second line. When RDBPRE creates a BASIC (BAS) source file from your input file (RBA), a header is printed on the second line of the BAS file that states the file name, time stamp, and the preprocessor version number. Forcing this text on the second line will cause BASIC compile-time errors when the first line is blank or continued. You can use a comment character as the first line of the program.

You cannot preprocess a program that attempts to access a non-existent database, unless your database refers to the data dictionary, CDD/Plus, and refers only to the definitions stored there. That is, if you specify a compile-time file name in the DATABASE statement, the database must exist at preprocess time. If you specify a compile-time path name in the DATABASE statement, the path name element must exist in the data dictionary at preprocess time. This is because the preprocessor must be able to validate relation and field definitions in the programs that refer to the database.

Use the exclamation point (!) as the comment character to put comments within Rdb/VMS data manipulation statements. Because the BASIC REM statement does not terminate the comment until the next line number, do not use the REM statement within Rdb/VMS data manipulation statements.

The preprocessor converts all Rdb/VMS data manipulation statements to comments in the BASIC program source file, replacing them with BASIC code. Therefore, if an Rdb/VMS data manipulation statement contains an exclamation point, the preprocessor doubles the exclamation point. For example, the statement &RDB& STORE P IN PORT USING P.CITY = "MY_CITY!" becomes:

```
!&RDB& STORE P IN PORT USING P.CITY = "MY_CITY!!" END_STORE
```

If the preprocessor did not double the exclamation point, the statement would look like the following in the BAS file:

```
!&RDB& STORE P IN PORT USING P.CITY = "MY_CITY!" END_STORE
```

The BASIC processor would interpret the following part of the text as a comment:

```
!&RDB& STORE P IN PORT USING P.CITY = "MY_CITY!
```

The remainder of the example, then, would be interpreted as a BASIC statement, and an error would result:

```
" END_STORE
```

13.2 Embedding DML Statements in the RDBPRE BASIC Program Environment

The Rdb/VMS data manipulation statements are a subset of the Relational Database Operator (RDO) utility statements. With the Rdb/VMS data manipulation statements, you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of the Rdb/VMS data manipulation statements.

13.2.1 Converting an RDO Prototype to the RDBPRE BASIC Environment

Once you have created a prototype of your queries in the interactive RDO facility, you are ready to convert these RDO statements to the BASIC program environment. See Chapter 7 for a full discussion of creating a prototype in RDO and for examples.

Example 13–1 is a BASIC subroutine based on the RDO prototype examples in Chapter 7.

Example 13–1 Converting an RDO Prototype to RDBPRE BASIC

```
Store_cand:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine stores a record in the CANDIDATES     !
! relation.  It shows how to store a value in a field   !
! of data type VARYING STRING.                          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PRINT FOR loop_cnt = 1% TO 24%
PRINT "Store Candidates"
PRINT
want_to_exit = 0%

Store_cand_1:

! Prompt user for data to store in the CANDIDATES relation.

UNTIL want_to_exit
        WHEN ERROR IN
              PRINT "Please enter the first name "+                   &
                              "of the candidate or press CTRL/Z: ";
              INPUT candidates::first_name
        USE
              want_to_exit = -1% IF ERR = 11%
        END WHEN
        EXIT STORE_CAND_1 IF want_to_exit
        confirm = 0%
        UNTIL confirm
              PRINT "Please enter the middle initial of the candidate: ";
              INPUT candidates::middle_initial
              PRINT "Please enter the last name of the candidate: ";
              INPUT candidates::last_name
              PRINT "Please enter candidate status information: ";
              INPUT  candidates::candidate_status::string_value
              PRINT "Have you entered the candidate"+                 &
                      " information correctly(Y/N): ";
              INPUT  answer
              confirm = -1% IF EDIT$(answer,32%) = "Y"
        NEXT
&RDB&   START_TRANSACTION READ_WRITE RESERVING CANDIDATES FOR SHARED WRITE
        success_flag = -1%

! Store the values specified by the user in the CANDIDATES
! relation.  Identify errors and inform user of the success or
! failure of the STORE operation.
```

(continued on next page)

Example 13–1 (Cont.) Converting an RDO Prototype to RDBPRE BASIC

```
&RDB&   STORE C IN CANDIDATES USING
&RDB&   ON ERROR
              success_flag  = 0%
              CALL Error_handler(RDB$STATUS, retry_count,             &
                  success_flag, lock_error)
&RDB&   END_ERROR
&RDB&         C.LAST_NAME   = candidates::last_name;
&RDB&         C.FIRST_NAME  = candidates::first_name;
&RDB&         C.MIDDLE_INITIAL = candidates::middle_initial;
&RDB&         C.CANDIDATE_STATUS = candidates::candidate_status::string_value
&RDB&   END_STORE
        IF success_flag
        THEN
              PRINT "Update operation succeeded"
&RDB&         COMMIT
        ELSE
              PRINT "Update operation failed"
&RDB&         ROLLBACK
        END IF
NEXT
RETURN ! To main module
```

The syntax you use for preprocessed Rdb/VMS data manipulation statements is not identical to the statement syntax you use in RDO. When you incorporate your prototype RDO statements into a program, you need to consider these areas:

- Use of host language variables
- Use of Rdb/VMS statement flags, described in Chapter 12.
- Differences in syntax
  - Using the GET statement instead of the PRINT statement
  - Nesting FETCH and GET operations within a host language loop
  - Using the ON ERROR and AT END clauses to detect error conditions
- Effects on structured programming
- Handling Rdb/VMS errors

13.2.1.1 Using Host Language Variables

A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. You can use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following statements allow the use of host language variables:

- Any data manipulation statement that permits the use of an RSE
- GET

- DATABASE (you can specify a database handle)
- READY
- FINISH

When you declare host language variables, simply follow the naming rules for BASIC. Ensure that host language variable data types and sizes are compatible with the corresponding database field data types and sizes. Refer to Chapter 8 for the list of equivalent BASIC data types.

Note that you cannot use the name of a database field (a context variable and a field name) as a subscript of an array.

Example 13–2 shows the use of host language variables to store a record. The host language variables appear in lowercase.

Example 13–2 Using Host Language Variables to Store a Record in RDBPRE BASIC

```
&RDB&  STORE J IN JOBS USING
&RDB&     J.JOB_CODE       = job_code;
&RDB&     J.JOB_TITLE      = job_title;
&RDB&     J.MAXIMUM_SALARY = max_sal;
&RDB&     J.MINIMUM_SALARY = min_sal;
&RDB&     J.WAGE_CLASS     = wage_class;
&RDB&  END_STORE
```

A convenient way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus. You can copy field and relation definitions, which include all the fields within the relation. However, you must be careful to copy only those field and relation definitions with data types that are supported by BASIC. See Chapter 12 for more information about using data dictionary definitions.

If your preprocessed BASIC program requires numeric data that is stored in the Rdb/VMS database as scaled values, you can place these scaled values in BASIC host language variables of any data type.

For example, suppose the following variables are defined in an Rdb/VMS database:

```
FIELD1          signed longword scale -2
FIELD2          signed longword
```

And suppose your BASIC program defines host language variables, HOST_VAR_1 and HOST_VAR_2, as packed decimal. In the following example, Rdb/VMS converts scaled numeric data to packed decimal when it places FIELD1 in HOST_VAR_1:

```
&RDB&   FOR R IN RELATION_X
&RDB&     GET
&RDB&       HOST_VAR_1 = R.FIELD1;
&RDB&       HOST_VAR_2 = R.FIELD2
&RDB&     END_GET
&RDB&   END_FOR
```

You can then use these host language variables to perform arithmetic operations in your BASIC program:

```
HOST_VAR = HOST_VAR_1 + HOST_VAR_2
```

However, you cannot perform arithmetic operations on scaled numeric data within an Rdb/VMS GET statement in BASIC programs. For example, if R.FIELD1 and R.FIELD2 are scaled numeric data, the following operation will fail:

```
&RDB&   FOR R IN RELATION_X
&RDB&    GET
&RDB&        HOST_VAR = R.FIELD1 + R.FIELD2
&RDB&    END_GET
&RDB&   END_FOR
```

Instead, to perform calculations on scaled numeric data, you must first put each database field value in a BASIC host language variable of any numeric data type. You can then use these host language variables in BASIC arithmetic expressions to perform the necessary calculations. In other words, store data in host language variables as scaled words, longwords, or quadwords, but manipulate them as if they were packed decimals.

13.2.1.2 Using Host Language Variables in Conditional Expressions You can use conditional expressions to limit the records included in a record stream. Conditional expressions contain one or more relational operators (see Table 3–1 in Section 3.5) and optionally logical operators (AND, OR, NOT).

In a programming environment, you probably do not want to code a specific value for the comparison string, as in:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING 'NH'
```

It is more likely that you want the user to supply the comparison string at run time. In this case, you need to declare a host language variable to hold the comparison string. For example:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING state_code
```

For the STARTING_WITH, MATCHING, and CONTAINING conditional expressions, you must declare your host language variable in such a way that the preprocessor can determine the correct length of the comparison string.

In BASIC, declare your host language variable as a string. The preprocessor will use the BASIC function LEN to determine the length of the varying string that is passed to the database. For example:

```
100  DECLARE STRING state_code, name, city

     ! Program statements
     ! Rdb/VMS statements: invoke database, start_transaction,
     ! and so on.

       .
       .
       .
&RDB& FOR E IN EMPLOYEES WITH
&RDB&   E.STATE MATCHING state_code
&RDB&   GET
&RDB&     name = E.LAST_NAME;
&RDB&     city = E.CITY;
&RDB&   END_GET
```

13.2.1.3 Converting DATE Data Type to TEXT DATE data types are stored in Rdb/VMS databases in encoded binary format. To display a date, you need to use the VMS system service routine, SYS$ASCTIM, to perform a conversion from encoded binary format to an ASCII string. First, place the binary value into a STRING host language variable that is mapped as two longwords. This gives you the required QUADWORD data type that the SYS$ASCTIM routine needs. Once you put the binary value into a host variable, you can convert it with SYS$ASCTIM to an ASCII string.

See the *VMS System Services Volume* for more information on using SYS$ASCTIM.

Note that RDBPRE uses the run-time library routine LIB$MOVC3 to move the value from the DATE data type to the host language variable. The preprocessor declares LIB$MOVC3 as external for you; do not declare it again in your program or you may receive a fatal compile-time error.

Example 13–3 is a code fragment from the ADD_EMPLOYEES subroutine that demonstrates how to display a date. In this example, the date is passed back and forth to Rdb/VMS with a string field in a RECORD structure. This method is used because this is the format you will get if you include the record definition from the data dictionary. This string is then moved into the WORK_DATE host language variable (an 8-byte string) that is also mapped as two longwords, WORK1 and WORK2. The WORK1 variable is then passed as the argument to SYS$ASCTIM.

Example 13–3 Using SYS$ASCTIM System Service Routine in RDBPRE BASIC

```
! Declare variables
EXTERNAL LONG FUNCTION SYS$ASCTIM,SYS$BINTIM

MAP (DATES)     STRING  work_date = 8%
MAP (DATES)     LONG    work1,work2


RECORD  EMPLOYEE
        STRING employee_id  = 5
        STRING last_name  = 14
        STRING first_name  = 10
        STRING middle_initial  = 1
        STRING address_data_1  = 25
        STRING address_data_2  = 25
        STRING city  = 20
        STRING state  = 2
        STRING postal_code  = 5
        STRING sex  = 1
        GROUP   birthday
                STRING string_value  = 8
        END GROUP
        STRING status_code  = 1
END RECORD
        .
        .
        .
              FOR i = 1 TO number_employees_added
&RDB&            FOR E IN EMPLOYEES WITH e.RDB$DB_KEY = database_key(i)
&RDB&               ON ERROR
                       success_flag = 0%
                       CALL Error_handler(RDB$STATUS,  &
                       retry_count, success_flag, lock_error)
&RDB&               END_ERROR
&RDB&               GET
&RDB&                  ON ERROR
                          success_flag = 0%
&RDB&                  END_ERROR
&RDB&                  employees::employee_id = E.EMPLOYEE_ID;
&RDB&                  employees::last_name   = E.LAST_NAME;
&RDB&                  employees::first_name  = E.FIRST_NAME;
&RDB&                  employees::middle_initial = E.MIDDLE_INITIAL;
&RDB&                  employees::address_data_1 = E.ADDRESS_DATA_1;
&RDB&                  employees::address_data_2 = E.ADDRESS_DATA_2;
&RDB&                  employees::city           = E.CITY;
&RDB&                  employees::state          = E.STATE;
&RDB&                  employees::postal_code    = E.POSTAL_CODE;
&RDB&                  employees::birthday::string_value = E.BIRTHDAY
&RDB&               END_GET
&RDB&             END_FOR
```

(continued on next page)

Example 13–3 (Cont.) Using SYS$ASCTIM System Service Routine in RDBPRE BASIC

```
! If the field values were successfully retrieved, then convert
! the date field from binary to a printable (ASCII) format.
! The first and last arguments to the call SYS$ASCTIM are not
! required arguments.

                     GOSUB Display_employee IF success_flag
                     success_flag = -1%
                     NEXT I
                        .
                        .
                        .
Display_employee:

PRINT
PRINT "Employee id: "+employees::employee_id
PRINT "Last name:   "+employees::last_name
PRINT "First name:  "+employees::first_name
PRINT "Middle init: "+employees::middle_initial
PRINT "Address:     "+employees::address_data_1+" "+            &
                        employees::address_data_2
PRINT "City:        "+employees::city
PRINT "State:       "+employees::state
PRINT "Postal code: "+employees::postal_code

!   Convert binary date to ASCII format.

work_date = employees::birthday::string_value
return_status = SYS$ASCTIM(,ascii_date,work1,)

PRINT "Birthday:    "+ascii_date
PRINT
RETURN
```

13.2.1.4 Converting ASCII DATE Strings to Binary Format

Use the VMS system service routine, SYS$BINTIM, to convert ASCII DATE strings into a binary representation so the DATE fields can be stored in the database.

See the *VMS System Services Volume* for more information on using SYS$BINTIM.

Example 13–4 is a code fragment from the ADD_EMPLOYEES subroutine that demonstrates how to use SYS$BINTIM in an RDBPRE BASIC program. In this example, the date is passed back and forth to Rdb/VMS with a string field in a RECORD structure. This method is used because this is the format you will get if you include the record definition from the data dictionary. This string is then moved into the WORK_DATE host language variable (an 8-byte string) that is also mapped as two longwords, WORK1 and WORK2. The WORK1 variable is then passed as the argument to SYS$BINTIM.

Example 13–4 Using SYS$BINTIM System Service Routine in RDBPRE BASIC

```
! Declare variables
EXTERNAL LONG FUNCTION SYS$ASCTIM,SYS$BINTIM

MAP (DATES)     STRING  work_date = 8%
MAP (DATES)     LONG    work1,work2

RECORD  EMPLOYEE
        STRING employee_id  = 5
        STRING last_name  = 14
        STRING first_name  = 10
        STRING middle_initial  = 1
        STRING address_data_1  = 25
        STRING address_data_2  = 25
        STRING city  = 20
        STRING state  = 2
        STRING postal_code  = 5
        STRING sex  = 1
        GROUP   birthday
                STRING string_value  = 8
        END GROUP
        STRING status_code  = 1
END RECORD
        .
        .
        .
! Prompt user to input date, keep prompting until user
! enters date in proper format.

UNTIL valid_date
   PRINT "Enter the Employee's birthday (dd-MMM-yyyy):";
      INPUT ascii_date
         ascii_date = EDIT$(ascii_date,32%)

         ! Use SYS$BINTIM to convert ASCII input to binary format.

         return_status = SYS$BINTIM(ascii_date,work1)
         IF (return_status AND 1%) <> 1%
         THEN
            PRINT "Invalid date format"
         ELSE
            valid_date = -1%
         END IF
NEXT
```

13.2.2 Using Literals

Use literal values to replace variables in the same way you would in any BASIC program. Literal values can be either numerics, character strings, or the generalized BASIC literal format (for example, B'01101'W, which indicates that the user wants a binary literal that is to be stored in a WORD). String literals must be quoted in double (" ") or single (' ') quotation marks in BASIC. You may use any literal in any Rdb/VMS data manipulation statement that accepts a host language variable.

```
&RDB&  FOR D IN DEPARTMENTS WITH
&RDB&    D.DEPARTMENT_CODE = "ADMN"
&RDB&    GET
&RDB&      DEP_NAME = D.DEPARTMENT_NAME
&RDB&    END_GET
&RDB&  END_FOR
```

13.2.3 Forming Record Streams

In BASIC, and any language that you use to access an Rdb/VMS database, you select the records you are interested in manipulating by gathering these records into a stream. You create this stream using the Rdb/VMS data manipulation statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

13.2.4 Retrieving Records

Rdb/VMS provides you with three statements to retrieve records:

- FOR
- Two START_STREAM statements:
  - Declared START_STREAM
  - Undeclared START_STREAM

The following sections provide BASIC examples of how to form record streams and retrieve records using the FOR and START_STREAM statements.

13.2.4.1 Using the FOR Statement to Retrieve Records The FOR statement forms a record stream and provides automatic iteration for any Rdb/VMS and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

Example 13–5 shows a FOR statement from the DISPLAY_CAND subroutine. It uses the flag “found_candidate_flag” to determine if the RSE has been satisfied. If a candidate record is found with field values that match the values in the host language variables, the success flag is set to true. If no record matches the values in the host language variables, then the success flag remains set to false.

Example 13–5 Using the FOR Statement in RDBPRE BASIC

```
             found_candidate_flag = 0%
&RDB&        FOR C IN CANDIDATES WITH
&RDB&           C.FIRST_NAME = candidates::first_name
&RDB&           AND
&RDB&           C.MIDDLE_INITIAL = candidates::middle_initial
&RDB&           AND
&RDB&           C.LAST_NAME = candidates::last_name

! Retrieve and display the VARYING STRING field if a record exists
! for the specified candidate.  If no record exists for this person,
! inform the user.

&RDB&           GET
&RDB&              candidates::candidate_status::string_value
&RDB&                         = C.CANDIDATE_STATUS
&RDB&           END_GET

                found_candidate_flag = -1%
                PRINT candidates::first_name+" "+              &
                      candidates::middle_initial+" "+ &
                      candidates::last_name+          &
                      " has the following status:"

                PRINT
                PRINT candidates::candidate_status::string_value
&RDB&        END_FOR
```

You can include host language statements within the FOR . . . END_FOR block to process the records within the stream. However, there is an important exception to the type of statement you can include. Do not transfer control out of the FOR . . . END_FOR block unless you do not want to return. It is impossible to enter the loop again once you have exited.

You may call a module from within a FOR loop because these subroutines execute within the FOR loop context. However, you cannot use a context variable defined in the FOR block in any subroutine that is preprocessed outside the FOR block.

13.2.4.2 Using Declared Streams to Retrieve Records Rdb/VMS supports two forms of the START_STREAM statement. The *declared* START_STREAM statement and the *undeclared* START_STREAM statement. Declared streams provide all the features of the undeclared streams and more. Most importantly, undeclared streams require that the statements you use to manipulate the stream be enclosed by the START_STREAM and END_STREAM statements in your source program. Declared streams do not impose this restriction. The statements you use to manipulate the stream may appear in any order within your program as long as the DECLARE_STREAM statement appears first and the statements execute in a logical order (START_STREAM, FETCH, GET, END_STREAM).

Digital recommends that all new applications use the declared START_STREAM statement. For this reason, only the declared START_STREAM statement is discussed in this section. Complete details on the differences between declared and undeclared START_STREAM statements are provided in Chapter 9.

Note *If you use the AT END clause in a FETCH statement, you must use the END_FETCH clause to terminate the FETCH statement. Do not use the BASIC REM statement or the BASIC line number within an AT END clause. These BASIC statements inadvertently terminate code generated by RDBPRE in the AT END clause. (Likewise, avoid these statements within FOR loops, ON ERROR statements, and nested constructs.)*

Example 13–6, from the sample program RDM$DEMO:B_SAMPLE.RBA, shows the use of the declared START_STREAM and FETCH statements. The example pairs a CANDIDATES record with an EMPLOYEES record at random. This could not be achieved with a FOR statement. You could not conditionally end a FOR loop when all the CANDIDATES records have been paired with EMPLOYEES records. A START_STREAM statement lets you do this.

Example 13–6 Using the Declared START_STREAM and FETCH Statements in RDBPRE BASIC

```
   .
   .
   .
! Declare streams used in the PAIR procedure

&RDB& DECLARE_STREAM cands USING CA IN CANDIDATES SORTED BY CA.LAST_NAME
&RDB& DECLARE_STREAM emps  USING EM IN EMPLOYEES  SORTED BY EM.FIRST_NAME
   .
   .
   .
Pair:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine demonstrates the use of the declared  !
! START_STREAM  statement.  The output of this program  !
! is merely a random matching of each CANDIDATES record !
! with an EMPLOYEES record.                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


&RDB& START_TRANSACTION READ_ONLY

! Open both streams and set a flag for the end of stream
! condition to false.

GOSUB Open_candidates
GOSUB Open_employees
end_of_emps = 0%
end_of_cands = 0%
```

(continued on next page)

Example 13–6 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE BASIC

```
! Fetch a record from the CANDIDATES and EMPLOYEES relations.

GOSUB Read_a_candidate
GOSUB Read_an_employee

! Print the employee and candidate names until the end-of-stream
! condition is met for the stream of CANDIDATES records.

UNTIL end_of_cands
        PRINT employees::last_name+" "+employees::first_name+   &
                '                    '+                         &
                candidates::last_name+" "+                      &
                candidates::first_name
        GOSUB Read_a_candidate
        IF NOT end_of_emps
        THEN
                GOSUB Read_an_employee
        END IF
NEXT

! Close both streams.

GOSUB Close_employees
GOSUB Close_candidates

&RDB& COMMIT.

PRINT "Press RETURN to continue ";
INPUT answer

RETURN ! To main module

! These subroutines control a stream.  Note that the statements
! do not appear in the order that they will be executed.  This
! is a feature that declared streams have and undeclared streams
! do not have.

Close_employees:

! Close the EMPLOYEES stream.

&RDB& END_STREAM emps.
RETURN

Close_candidates:

! Close the CANDIDATES stream.

&RDB& END_STREAM cands.
RETURN

Open_candidates:

! Open the CANDIDATES stream.

&RDB& START_STREAM cands.
RETURN

Open_employees:

! Open the EMPLOYEES stream.
```

(continued on next page)

Example 13–6 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE BASIC

```
&RDB& START_STREAM emps.
RETURN

Read_a_candidate:

! Fetch a CANDIDATES record.

&RDB& FETCH cands
&RDB&   AT END
                end_of_cands = -1%
&RDB& END_FETCH
        IF NOT end_of_cands
        THEN
&RDB&           GET
&RDB&                   candidates::last_name   = CA.LAST_NAME;
&RDB&                   candidates::first_name  = CA.FIRST_NAME;
&RDB&                   candidates::candidate_status::string_value
&RDB&                                   = CA.CANDIDATE_STATUS
&RDB&           END_GET
        END IF
RETURN
Read_an_employee:

! Fetch an EMPLOYEES record.

&RDB& FETCH emps
&RDB&   AT END
                end_of_emps = -1%
&RDB& END_FETCH
        IF NOT end_of_emps
        THEN
&RDB&           GET
&RDB&                   employees::last_name = EM.LAST_NAME;
&RDB&                   employees::first_name = EM.FIRST_NAME;
&RDB&                   employees::employee_id = EM.EMPLOYEE_ID
&RDB&           END_GET
        END IF
RETURN   ! To main module
```

13.2.5 Retrieving Segmented Strings

Retrieving segmented strings is a two-step process. First, you must retrieve the record that contains the segmented string field; then, you must retrieve the individual segments that comprise the segmented string field.

You may find it easier to picture a segmented string by referring to Figure 8–1 in Chapter 8.

Rdb/VMS provides you with two statements to retrieve segmented string fields:

- FOR
- START_SEGMENTED_STRING

When you retrieve segmented strings in RDBPRE BASIC programs, you must use a static string descriptor to receive the segmented string segments, rather than accept the BASIC default dynamic class descriptor. If you accept the dynamic class descriptor, Rdb/VMS may write over portions of your program's data.

13.2.5.1 Using the FOR Statement to Retrieve Segmented Strings You must use two streams when processing segmented string streams. Use the first FOR or START_STREAM statement to form an outer stream of records, and then use the second FOR statement to form an inner stream of segments. This inner stream formed by the second RSE identifies the segments contained in the field specified by the outer stream formed by the first RSE. Use different context variables in the inner and outer streams.

Remember that to retrieve the segmented string, you must begin at the first segment and retrieve segments in the order that they are stored, that is, sequentially.

Example 13–7, from the DISPLAY_RESUME subroutine:

- Uses a FOR statement to search the database for a record with a value for the EMPLOYEE_ID field that matches the host language variable, employees::employee_id
- Uses a second FOR statement to loop through the segments of the segmented string field for the EMPLOYEES record
- Uses the GET statement to retrieve the individual segments that comprise a segmented string
- Displays these values on the terminal

Example 13–7 Using the FOR Statement with Segmented Strings in RDBPRE BASIC

```
&RDB&   START_TRANSACTION READ_ONLY
        found_employee_flag = 0%

! Start an outer FOR loop to retrieve the employee record(s)
! with the specified ID.

&RDB&   FOR R IN RESUMES WITH R.EMPLOYEE_ID = employees::employee_id
                found_employee_flag = -1%

! Start an inner FOR loop to retrieve the segments
! of the segmented string that comprise the employee’s
! resume.

&RDB&           FOR RR IN R.RESUME
&RDB&           GET
&RDB&                   resume_segment = RR.RDB$VALUE;
&RDB&                   segment_length = RR.RDB$LENGTH
&RDB&           END_GET
! Display each segment as it is retrieved from the database.

                PRINT LEFT(resume_segment,segment_length)
&RDB&           END_FOR
&RDB&   END_FOR
&RDB&   COMMIT

! If a record with the specified ID was not found then inform
! the user.

        IF NOT found_employee_flag
        THEN
                PRINT ’Employee: ’, employees::employee_id,     &
                                ’ has no resume on file’
        END IF
```

The GET statement fetches only as much of the stored segment as the host language variable that receives the segment can hold. The next GET statement fetches the next piece of the segment. Suppose the segmented string segment size in the previous example was declared as 80 characters and the actual length of the stored segment was 100 characters. The first GET statement would fetch 80 characters of the first segment and the next GET statement would fetch the remaining 20 characters. The third GET statement would fetch 80 characters of the second segment, the next GET statement would fetch the remaining 20, and so on.

13.2.5.2 Using the START_SEGMENTED_STRING Statement to Retrieve Segmented Strings

When you want to maintain program control of loop iteration, use the START_SEGMENTED_STRING statement with a record stream formed by a FOR or START_STREAM statement. You must start two streams when processing segmented string streams with the START_SEGMENTED_STRING statement.

Form an outer stream of records with a FOR or START_STREAM statement, then use the START_SEGMENTED_STRING statement to form an inner stream of segments. This inner stream identifies the segment stream that is contained in the field specified by the outer FOR or START_STREAM statement. When you name the segment stream, use a different name from the name used in the outer stream. Also, use different context variables for the outer stream and the inner segmented string stream.

The program shown in Example 13–8:

- Uses an undeclared START_STREAM statement to find all the records in the RESUMES relation with an employee ID of 12345.
- Uses a START_SEGMENTED_STRING statement to retrieve the resume of each employee record found by the first stream.
- Uses the GET statement to retrieve the segments that comprise the segmented string.
- Checks the return status value of the GET statement after each segment is retrieved to make sure the end-of-segmented-string condition has not been met. If this condition has not been met, the value of the current segment is printed.
- Stops processing the segmented string field when the preceding condition is met.
- Fetches the next employee record with an employee ID of 12345, if one exists.
- Closes both streams when both the START_STREAM and START_SEGMENTED_STRING end conditions have been met.
- Commits the transaction.

Example 13–8 Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE BASIC

```
MAP(RESUMES) STRING resume_segment = 80%
EXTERNAL LONG CONSTANT RDB$_SEGSTR_EOF

&RDB&   DATABASE pers = FILENAME 'MF_PERSONNEL'


&RDB&   START_TRANSACTION READ_ONLY

! Find all the records in the RESUMES relation
! with an employee ID of 12345.

&RDB&   START_STREAM RESSTR USING
&RDB&           R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
&RDB&                FETCH RESSTR
&RDB&                END_FETCH

! Retrieve the segments that comprise the segmented string field.

&RDB&   START_SEGMENTED_STRING RINFO USING STRN IN R.RESUME
        end_of_stream = -1%
WHILE end_of_stream

&RDB&           GET
&RDB&             ON ERROR
&RDB&             END_ERROR
&RDB&                   resume_segment = STRN.RDB$VALUE;
&RDB&                   segment_length = STRN.RDB$LENGTH
&RDB&           END_GET

! Check the return status of the GET statement after each segment
! is retrieved to make certain that the end-of-segmented-string
! condition has not been met.  If this condition has not been met,
! print the value of the current segment.  Otherwise, stop processing
! the stream of segments.

                IF RDB$LU_STATUS <> RDB$_SEGSTR_EOF
                THEN
                        PRINT LEFT(resume_segment,segment_length)
                ELSE
                        end_of_stream = 0%
                END IF
NEXT

! Close both streams.

&RDB&   END_SEGMENTED_STRING RINFO
&RDB&   END_STREAM RESSTR
&RDB&   COMMIT

&RDB& FINISH
EXIT PROGRAM
```

13.2.6 Retrieving Field Values

Use the GET statement to retrieve one, several, or all the field values from a database record. You can also use the GET statement to retrieve statistical values from the database.

Do not use the RDBPRE concatenation operator (|) in a GET statement. Doing so causes a preprocessing error. To concatenate fields in preprocessed programs, first use the GET statement to retrieve the individual fields and store them in separate BASIC variables. Then concatenate the BASIC variables in a BASIC statement using the BASIC concatenation operator, the plus sign (+).

Section 13.2.6.1 and Section 13.2.6.2 provide examples of retrieving field and record values. Section 13.2.6.3 provides an example of retrieving statistical values.

13.2.6.1 Using the GET Statement to Retrieve Field Values When you form a record stream using the FOR statement, you include the GET statement within the FOR . . . END_FOR block to retrieve field values from the record stream. When you form a record stream using the undeclared START_STREAM statement, you include the GET statement between the START_STREAM and END_STREAM statements. When you use the declared form of the START_STREAM statement, the GET statement must execute within the START_STREAM . . . END_STREAM block, however, it does not have to appear within this block in your program.

Example 13–9, from the LIST_RECORD subroutine, shows the use of the FOR and GET statements in RDBPRE BASIC.

Example 13–9 Using the FOR and GET Statements in RDBPRE BASIC

```
&RDB& FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
&RDB&   FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&           GET
&RDB&                   employees::first_name = E.FIRST_NAME;
&RDB&                   employees::last_name  = E.LAST_NAME;
&RDB&                   degrees::degree       = D.DEGREE;
&RDB&                   degrees::degree_field = D.DEGREE_FIELD
&RDB&           END_GET
                PRINT "Name is: "+employees::first_name+" "+    &
                       employees::last_name+" "+                &
                      "Degree is: "+degrees::degree+" "+        &
                      "Degree field is: "+degrees::degree_field
&RDB&   END_FOR
        .
        .
        .
&RDB&END_FOR
```

See Example 13–6 for a demonstration of how to use the START_STREAM, FETCH, and GET statements.

13.2.6.2 Using the GET * Statement to Retrieve Field Values A special form of the GET statement is the GET * statement, which lets you retrieve database values at the record level rather than the field level. You can retrieve all the fields in a record with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the records of a relation, with record field names that match the database field names. You can use the BASIC %INCLUDE %FROM %CDD directive to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) The GET * statement in the following example retrieves all the fields from the records of the JOB_ HISTORY relation and places their values in the job_history host language record structure:

```
&RDB& FOR FIRST 1 J IN JOB_HISTORY WITH
&RDB&   J.JOB_CODE = JOB_CODE IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   GET
&RDB&     job_history = J.*
&RDB&   END_GET
&RDB& END_FOR
```

13.2.6.3 Using the GET Statement to Retrieve Statistical Values

You can retrieve the result of a statistical expression directly, without processing each record in the record stream. RDBPRE may assign a data type to the result that is different from the data type of the field referred to in the expression. See Chapter 8 for information on the data type conversions performed by statistical expressions.

Example 13–10, from the STATS subroutine, uses the COUNT statistical function to find the total number of records in the EMPLOYEES relation.

Example 13–10 Using the GET Statement to Retrieve a Statistical Value in RDBPRE BASIC

```
Stats:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine displays the total number of records stored in the !
! EMPLOYEES relation.                                                !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PRINT FOR loop_cnt = 1% TO 24%
PRINT "Statistics"
PRINT
&RDB& START_TRANSACTION READ_ONLY

        PRINT "The number of employees in the Corporation is: ";

! Use the GET statement with a statistical function to calculate
! the total number of records in the EMPLOYEES relation.

&RDB& GET number_of_employees = COUNT OF E IN EMPLOYEES END_GET

! Display the value.

        PRINT number_of_employees
&RDB& COMMIT

PRINT
PRINT "Press RETURN to continue ";
INPUT answer
```

13.2.7 Updating Records Using the STORE, MODIFY, and ERASE Statements

The Rdb/VMS update statements can only be used in a read/write transaction. (You may, of course, include any valid Rdb/VMS statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

If you update a record and triggered actions have been defined for the relation containing the record, the update operation (STORE, MODIFY, or ERASE) will have the specified effect on all the relations in the database that have a foreign key relationship with the record you want to update.

If a relation-specific constraint has been defined, your ability to perform update operations may depend on the presence of matching field values in other relations. For more information on relation-specific constraints, see Section 6.6.

Include the GET statement in a read/write transaction if you intend to update any of the fields returned by the GET statement.

Note *You may not use a view to update records if that view refers to more than one relation.*

13.2.7.1 Storing Records You can insert values in one or more fields in one record using a single STORE statement. To store more than one record in a relation, include the STORE statement within a program loop.

Example 13–11, from the ADD_EMPLOYEES subroutine, stores an employee record in the EMPLOYEES relation.

Example 13–11 Storing Records in RDBPRE BASIC

```
        .
        .
        .
! User entered the values to be stored in the database earlier
! in program.

&RDB&    STORE E IN EMPLOYEES USING
&RDB&       ON ERROR
               success_flag = 0%
               CALL ERROR_HANDLER(RDB$STATUS,          &
                                  retry_count,            &
                                  success_flag,           &
                                  lock_error)
&RDB&       END_ERROR

! Store the values that the user entered in an EMPLOYEES record.
```

(continued on next page)

Example 13–11 (Cont.) Storing Records in RDBPRE BASIC

```
&RDB&       E.EMPLOYEE_ID    = employees::employee_id;
&RDB&       E.LAST_NAME      = employees::last_name;
&RDB&       E.FIRST_NAME     = employees::first_name;
&RDB&       E.MIDDLE_INITIAL = employees::middle_initial;
&RDB&       E.ADDRESS_DATA_1 = employees::address_data_1;
&RDB&       E.ADDRESS_DATA_2 = employees::address_data_2;
&RDB&       E.CITY           = employees::city;
&RDB&       E.STATE          = employees::state;
&RDB&       E.POSTAL_CODE    = employees::postal_code;
&RDB&       E.BIRTHDAY       = employees::birthday::string_value
               .
               .
               .
&RDB&    END_STORE
```

13.2.7.1.1 Using the STORE * Statement to Store Records A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in a record with the STORE * statement. To use the STORE * statement, you must first declare a record structure that contains all the fields in the relation, with record field names that match the database field names. You can use the BASIC %INCLUDE %FROM %CDD directive to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to store in the record fields and store the entire record using the STORE * statement. Example 13–12 shows the use of the STORE * statement to store a host language record, job_history in the JOB_HISTORY relation.

Example 13–12 Using the STORE * Statement in RDBPRE BASIC

```
&RDB& STORE J IN PERS.JOB_HISTORY USING
&RDB&   J.* = job_history
&RDB& END_STORE
```

13.2.7.1.2 Using the CREATE_SEGMENTED_STRING Statement to Store Segmented Strings Use the CREATE_SEGMENTED_STRING statement and the STORE statement to store segmented strings in a relation. You must use two operations to store segmented strings.

Note *See Section 9.2.6.1.2 for information about defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name with an appropriate value for storing your segmented strings.*

Note *Segmented strings cannot be updated (ERASE, MODIFY, or STORE) as part of a triggered action. For more information, see the DEFINE TRIGGER statement in the* VAX Rdb/VMS RDO and RMU Reference Manual.

Example 13–13, from the MOD_RESUME subroutine, demonstrates how to read and store a resume into a segmented string from a sequential file; then it shows how to use the segmented string handle to modify an existing database record.

Example 13–13 Using the CREATE_SEGMENTED_STRING Statement in RDBPRE BASIC

```
Mod_resume:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine demonstrates how to modify a  !
! field of data type SEGMENTED STRING.          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PRINT FOR loop_cnt = 1% TO 24%
PRINT  "Modify a resume"
PRINT

want_to_exit = 0%
Mod_resume_1:

! Prompt user for the employee ID of the RESUMES record
! he or she wants to modify.

UNTIL want_to_exit
        WHEN ERROR IN
                PRINT "Please enter the ID of the employee or press CTRL/Z: ";
                INPUT employees::employee_id
        USE
                want_to_exit = -1%
        END WHEN
        EXIT Mod_resume_1 IF want_to_exit
! Prompt user for the file name of the resume that will replace
! the old resume.

        PRINT "To modify a resume, you must supply a new resume"
        PRINT " to replace the old resume"
        PRINT
        file_ok = 0%
        UNTIL file_ok
                file_ok = -1%
                PRINT "Please enter file name of new resume: ";
                INPUT resume_file
                WHEN ERROR IN
                        OPEN resume_file FOR INPUT AS FILE 1%
                USE
                        PRINT "File - ";resume_file;" - not found"
                        file_ok = 0%
                END WHEN
        NEXT
```

(continued on next page)

Example 13–13 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in RDBPRE BASIC

```
&RDB&   START_TRANSACTION READ_WRITE RESERVING RESUMES FOR SHARED WRITE

! Create a new segmented string that will hold the value
! of the new resume.

&RDB&   CREATE_SEGMENTED_STRING resume_handle

        eof_flag = 0%

Resume_read:

        UNTIL eof_flag
                resume_line = ""
                WHEN ERROR IN
                        INPUT LINE #1%, resume_line
                USE
                        eof_flag = -1%
                END WHEN
                EXIT Resume_read IF eof_flag
                resume_line = EDIT$(resume_line,4%)

&RDB&           STORE R IN resume_handle USING R.RDB$VALUE =
&RDB&                           resume_line END_STORE
        NEXT

        CLOSE #1%
&RDB&   END_SEGMENTED_STRING resume_handle
! Modify the old resume by supplying the segmented
! string handle from the CREATE_SEGMENTED_STRING
! statement as the object of the segmented string
! assignment statement.

&RDB&   FOR R IN RESUMES WITH R.EMPLOYEE_ID = employees::employee_id
&RDB&      MODIFY R USING
&RDB&           R.RESUME      = resume_handle
&RDB&      END_MODIFY
&RDB&   END_FOR
&RDB&   COMMIT

NEXT
RETURN ! to main module
```

13.2.7.2 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of a record in a relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Example 13–14, a BASIC program segment from the MODIFY_ADDRESS subroutine, modifies a record in the EMPLOYEES relation. The values used to modify the record were requested earlier in the program.

Example 13–14 Modifying Records in RDBPRE BASIC

```
                               .
                               .
                               .
&RDB&   START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR SHARED WRITE

! Modify the address fields for the specified EMPLOYEES record.

&RDB&   FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employees::employee_id
&RDB&           MODIFY E USING
&RDB&           ON ERROR
                        success_flag = 0%
                        CALL Error_handler (RDB$STATUS,         &
                                                retry_count,    &
                                                success_flag,   &
                                                lock_error)
&RDB&           END_ERROR
&RDB&                   E.ADDRESS_DATA_1 = employees::address_data_1;
&RDB&                   E.ADDRESS_DATA_2 = employees::address_data_2;
&RDB&                   E.CITY           = employees::city;
&RDB&                   E.STATE          = employees::state;
&RDB&                   E.POSTAL_CODE    = employees::postal_code;
&RDB&           END_MODIFY
&RDB&   END_FOR
        IF success_flag

! Notify the user of the success or failure of the modify operation.

        THEN
                PRINT "Update operation succeeded"
&RDB&           COMMIT
        ELSE
                PRINT "Update operation failed"
&RDB&           ROLLBACK
        END IF
```

13.2.7.2.1 Using the MODIFY * Statement to Modify Records A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. You can use the BASIC %INCLUDE %FROM %CDD statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement.

Only use the MODIFY * statement if you need to modify every field value in a record. Modifying a field by replacing one value with an identical value needlessly adds overhead to your program. For example, your program may check constraints on a field value that *you know* is valid because it is the same value that the field presently holds.

Example 13–15 replaces the field values of an employee record in the JOB_HISTORY relation with the values in the job_history host language record structure.

Example 13–15 Using the MODIFY * Statement in RDBPRE BASIC

```
&RDB& FOR J IN JOB_HISTORY WITH
&RDB&   J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   MODIFY J USING
&RDB&     J.* = job_history
&RDB&   END_MODIFY
&RDB& END_FOR
```

13.2.7.2.2 Modifying Segmented Strings To modify a segmented string, you must first create a new segmented string with the CREATE_SEGMENTED_STRING statement and then modify the existing record by replacing the logical pointer to the old segmented string with the logical pointer to the new segmented string. You accomplish this by using the segmented string handle in an assignment statement. As Chapter 8 explains in more detail, when you store a segmented string field, you do not actually store segments into a record—you store a logical pointer to the first segment in the segmented string. Thus, by creating a new segmented string and a new segmented string id associated with it, you can modify the field in a database record that "contains" a segmented string merely by replacing the old segmented string id with a new segmented string id. When you use the segmented string handle in an assignment statement, RDBPRE understands that it is the segmented string id which is to be assigned to the record.

Note *Although you use a MODIFY statement to modify segmented strings, you are not actually modifying the individual segments that comprise the segmented string field. You are actually replacing the entire segmented string with a new segmented string.*

See an earlier example, Example 13–13, for an illustration of how this is done in BASIC.

13.2.7.3 Erasing Records You can delete one, many, or all the records from a relation using a single ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase.

Example 13–16, from the DELETE_RECORD subroutine, demonstrates how to ERASE records in BASIC programs.

Note *The definition of the sample personnel database includes the trigger EMPLOYEE_ID_CASCADE_DELETE, which performs an automatic deletion of records in the relations named in ERASE statements in Example 13–16 (except for RESUMES) when the record with the matching employee ID is deleted from the EMPLOYEES relation. Thus, you would not need to include "cascading deletion" logic in your programs if it were already included in a trigger definition.*

Example 13–16 Erasing Records in RDBPRE BASIC

```
! Earlier in the subroutine DELETE_RECORD, an employee was retrieved
! to make certain that the user wants to delete this employee's
! records.  Having made that determination, the program will now
! delete all records associated with that employee.  When the
! employee record was retrieved, the database key associated with
! that record was also retrieved.  It can be used here to quickly
! locate that employee's EMPLOYEES record again, so that records for
! this employee can be erased from all the relations in which he or
! she has a record.

&RDB&   START_TRANSACTION READ_WRITE RESERVING EMPLOYEES,
&RDB&        SALARY_HISTORY, JOB_HISTORY, DEPARTMENTS,
&RDB&        DEGREES, WORK_STATUS, RESUMES FOR SHARED WRITE

&RDB&   FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = db_key
&RDB&        FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                ERASE JH
&RDB&        END_FOR
&RDB&        FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                ERASE SH
&RDB&        END_FOR
&RDB&        FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                ERASE D
&RDB&        END_FOR
&RDB&        FOR R IN RESUMES WITH R.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                ERASE R
&RDB&        END_FOR
&RDB&        ERASE E
             PRINT "Employee id: "+employees::employee_id+          &
                             " deleted successfully"
&RDB&   END_FOR
```

13.3 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer or address that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this key as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, the scope of a dbkey ends with the COMMIT statement. That is, a dbkey is guaranteed to point to the same record for the life of the transaction in which it is retrieved.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

The following example demonstrates how to specify the dbkey scope in an RDBPRE BASIC program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH
```

Suggestions on how you can take advantage of the dbkey scope are contained in Section 9.2.7.

13.4 Using Structured Programming

Programs and modules that pass through the RDBPRE preprocessor do not have unlimited freedom in structure. Calls to routines, such as the BASIC GOSUB block, or calls to subprograms and subroutines require that you pay special attention to the context from which they are called.

Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- DECLARE_STREAM
- FOR
- GET
- START_STREAM
- END_STREAM
- FETCH
- STORE

- MODIFY
- ERASE
- CREATE_SEGMENTED_STRING
- START_SEGMENTED_STRING
- END_SEGMENTED_STRING

These individual data manipulation statements each form only part of a complex call to the database. The preprocessor generates one call to the database, using more than one data manipulation statement. For example, a MODIFY statement executes within the context of a FOR or START_STREAM statement. The call to the database can only be made using both the FOR and MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. However, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

Also keep in mind that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement, and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which it is issued.

A stream declared with the DECLARE_STREAM statement lets you place the stream of manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

For more information on the declared and undeclared START_STREAM statement, see Section 9.2.3.2. Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- DATABASE
- READY
- START_TRANSACTION
- GET

- COMMIT
- ROLLBACK
- FINISH
- DECLARE_STREAM

Remember that you must issue the DECLARE_STREAM statement before you can issue a declared START_STREAM statement, and the DATABASE statement must appear in the data declaration section of your program.

Example 13–17, from the DELETE_RECORD and CALL_OTHER subroutines, demonstrates structured programming in a preprocessed BASIC program. The DELETE_RECORD module and the CALL_OTHER subroutine are separately preprocessed and compiled. They are linked with the LINK command. The DELETE_RECORD module passes the value of the dbkey to the CALL_OTHER subroutine. This subroutine finds the record associated with the dbkey and displays this record on the terminal. Although it is not necessary to program this query in two modules, it is done here to demonstrate how to pass variables between separately processed modules.

Example 13–17 Using Data Manipulation Statements in Structured Programming in RDBPRE BASIC

```
Subroutine DELETE_RECORD:
   .
   .
   .
        confirm = 0%
        success_flag = -1%
        until confirm OR want_to_exit
        trans1 = 0%
&RDB&           START_TRANSACTION (TRANSACTION_HANDLE trans1)
&RDB&                   READ_WRITE RESERVING EMPLOYEES FOR SHARED READ
                found_employee_flag = 0%

! Find the employee record that the user wants to delete.  If
! an error occurs during the FOR operation, call an error handler.

&RDB&              FOR (TRANSACTION_HANDLE trans1)
&RDB&                  E IN EMPLOYEES WITH
&RDB&                    E.EMPLOYEE_ID = employees::employee_id
&RDB&                 ON ERROR
                         success_flag = 0%
                         CALL Error_handler(RDB$STATUS,          &
                                            retry_count,    &
                                           success_flag,   &
                                           lock_error)
&RDB&                 END_ERROR
```

(continued on next page)

Example 13–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE BASIC

```
! Get the dbkey of the EMPLOYEES record that the user wants to delete.

&RDB&                 GET
&RDB&                   ON ERROR
                           success_flag = 0%
&RDB&                   END_ERROR
&RDB&                   db_key = E.RDB$DB_KEY
&RDB&                 END_GET
                      found_employee_flag = -1%
&RDB&              END_FOR
                   IF NOT found_employee_flag
                   THEN
                      PRINT "No employee with id: "+          &
                            employees::employee_id+" on file"
                   ELSE
! Pass the dbkey to an external routine CALL_OTHER to
! print out the record to which the dbkey points.  Note
! that using an external routine is neither necessary nor recommended
! for performing this task.  It is done in this example only to show
! how values are passed between routines in an RDBPRE BASIC program.

                       IF success_flag
                          THEN CALL Call_other(db_key, trans1)
                       END IF
                   END IF
&RDB&            COMMIT (TRANSACTION_HANDLE trans1)

! Ask user for confirmation that this is the EMPLOYEES
! record he or she wants to delete.

                 PRINT
                 IF found_employee_flag
                 THEN
                    PRINT "Is this the employee you want to delete (Y/N): ";
                    INPUT answer
                    confirm  = -1% IF EDIT$(answer,32%) = "Y"
                 END IF

                 IF NOT confirm
                 THEN
                    PRINT "Employee with employee id: "+            &
                    employees::employee_id+" not deleted"
                    PRINT
                 END IF
                   .
                   .
                   .
```

(continued on next page)

Example 13–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE BASIC

Subroutine CALL_OTHER:

```
SUB CALL_OTHER(STRING db_key, LONG trans_1)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine is passed the dbkey and transaction handle              !
! from the DELETE_RECORD subroutine in the program B_SAMPLE.RBA.          !
! With this information, the program can find and display                 !
! the employee record associated with an employee_id specified in         !
! DELETE_RECORD and then return program control to the DELETE_RECORD      !
! subroutine.                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
RECORD EMPLOYEE
        STRING employee_id = 5
        STRING last_name = 14
        STRING first_name = 10
        STRING middle_initial = 1
        STRING address_data_1 = 25
        STRING address_data_2 = 25
        STRING city = 20
        STRING state = 2
        STRING postal_code = 5
        STRING sex = 1
        GROUP birthday
                STRING string_value = 8
        END GROUP
        STRING status_code = 1
END RECORD

MAP (RECORDS)   employee                employees

! Because the database was invoked in the main program
! with GLOBAL attributes, refer to it here as EXTERNAL.

&RDB&  DATABASE EXTERNAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH

! The transaction was started in the DELETE_RECORD subroutine,
! so there is no need to start a transaction here.  Use the
! transaction handle to identify the request with the transaction
! started in the DELETE_RECORD subroutine.  Use the dbkey found in
! DELETE_RECORD to locate the correct employee record.

&RDB&   FOR (TRANSACTION_HANDLE trans_1) E IN EMPLOYEES WITH
&RDB&           E.RDB$DB_KEY = db_key
&RDB&           GET
&RDB&                   employees::employee_id = E.EMPLOYEE_ID;
&RDB&                   employees::last_name = E.LAST_NAME;
&RDB&                   employees::first_name = E.FIRST_NAME;
&RDB&                   employees::middle_initial = E.MIDDLE_INITIAL;
&RDB&                   employees::address_data_1 = E.ADDRESS_DATA_1;
&RDB&                   employees::address_data_2 = E.ADDRESS_DATA_2;
&RDB&                   employees::city = E.CITY;
&RDB&                   employees::state = E.STATE;
&RDB&                   employees::postal_code = E.POSTAL_CODE;
&RDB&                   employees::birthday::string_value = E.BIRTHDAY
&RDB&           END_GET
```

(continued on next page)

Example 13–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE BASIC

```
! Display the EMPLOYEES record.
      PRINT
      PRINT "Employee id: ", employees::employee_id
      PRINT "Last name: ", employees::last_name
      PRINT "First name:", employees::first_name
      PRINT "Middle init: ", employees::middle_initial
      PRINT "Address: ", employees::address_data_1, employees::address_data_2
      PRINT "City:", employees::city
      PRINT "State:", employees::state
      PRINT "Postal code: ", employees::postal_code
&RDB&   END_FOR
END SUB
```

13.4.1 Using Handles in Structured Programming

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions
- Requests

Information on when and how to use request handles is supplied in Chapter 9. Section 13.4.2 and Section 13.4.4 discuss how to declare handles in an RDBPRE BASIC program.

13.4.2 Declaring and Initializing Handles

With the exception of the database handle, declaring handles in RDBPRE BASIC is similar to declaring any other program variable. The declaration and initialization of a database handle is done simply by specifying the handle in the DATABASE statement. You do not declare a database handle in the data declaration portion of your BASIC program. RDBPRE initializes the handle for you. You should not assign a value to a database handle with an assignment statement (or any other way).

User-specified request and transaction handles must be declared in the data declaration portion of your program. In BASIC, declare user-specified request and transaction handles as longwords and initialize them to zero.

If you want to release the resources associated with a request handle, you can do so by issuing a FINISH statement, or, if you do not want to detach from the database, you can release the request by issuing a call to the RDB$RELEASE_REQUEST procedure with the following statement (where req1 is a user-supplied request handle):

```
return_stat = (RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, req1)
    IF (return_stat AND 1%) = 0% THEN
          CALL SYS$PUTMSG(RDB$MESSAGE_VECTOR)
    END IF
```

Declare the variable that holds the return status value as LONG and RDB$RELEASE_REQUEST as EXTERNAL LONG.

13.4.3 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

13.4.4 Declaring and Initializing Distributed Transaction Identifiers

Declaring distributed transaction identifiers in RDBPRE BASIC is similar to declaring any other program variable. Distributed transaction identifiers must be declared in the data declaration portion of your BASIC program. Declare a distributed transaction identifier as two longwords and initialize it to zero. You should not assign a value to a distributed transaction identifier with an assignment statement.

13.5 Using Callable RDO

The RDBPRE preprocessor statements do not include data definition statements. If you want to perform data definition within your preprocessed program, you must use the Callable RDO program interface. For example, during a batch process, or when others are not using the database, your program may define a temporary index on a field to facilitate Rdb/VMS performance during your program execution.

You can also use Callable RDO when your program needs the ability to form dynamic queries. That is, when your program will not know what a query is until run time. Otherwise, you should use the RDBPRE preprocessor when possible for all BASIC data manipulation operations. Preprocessed Rdb/VMS statements execute significantly faster than calls using the function RDB$INTERPRET.

When using Callable RDO, your program communicates with Rdb/VMS using the RDB$INTERPRET function. You call RDB$INTERPRET to pass your data manipulation or data definition statement to Rdb/VMS. Declare RDB$INTERPRET as an integer (longword) function. The RDB$INTERPRET function returns a status value that indicates the success or failure of the function. The return status value is a systemwide condition value that indicates either success or a unique Rdb/VMS symbolic error code. Your program declares a longword variable to hold the return status value so you can test the success or failure of the call. (Refer to Chapter 10 and Section 13.6 in this chapter for further information on handling Rdb/VMS run-time exception conditions.)

The BASIC format of the RDB$INTERPRET calling sequence is:

```
ret-stat = RDB$INTERPRET('rdb-statement'[, host-var [BY DESC] ,...])
```

The arguments for the RDB$INTERPRET function are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement

  The Rdb/VMS statement you pass to Rdb/VMS. Handle rdb-statement according to your language's rules for handling string literals or string variables.

- host-var

  A host language variable you pass to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*. You must include a by-descriptor passing mechanism when your language's default passing mechanism for the host language variable data type is not by descriptor. Refer to the BASIC language reference manual for the specific format of the passing mechanism.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some Rdb/VMS statements may produce unwieldy code in the call to the RDB$INTERPRET function. Instead, assign the Rdb/VMS statement string literal to a string variable. Then pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your Rdb/VMS data manipulation statements from the mechanics of using the RDB$INTERPRET function.

Callable RDO program development is explained in detail in Chapter 19.

The following section discusses the use of the DATABASE statement and the visibility of transactions in preprocessed programs that use Callable RDO.

13.5.1 Using the DATABASE Statement with Embedded Callable RDO

You must use an INVOKE DATABASE statement in your preprocessed RDBPRE program and a separate RDO INVOKE DATABASE statement in the embedded Callable RDO statements. To ensure that the preprocessor invokes the identical database for the preprocessed and the Callable RDO portions of the program, use the same database handle in each INVOKE DATABASE statement. Invoke the database:

- In the preprocessed program, using a GLOBAL or EXTERNAL database handle.
- In the Callable RDO program, by passing the database handle to the RDB$INTERPRET function.

For more information on database handles, see the section on handles in Chapter 9.

In Callable RDO, you must pass the database handle to RDB$INTERPRET as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both RDBPRE and Callable RDO INVOKE DATABASE statements in the same program module. The preprocessor ignores any statement that is not preceded by the Rdb/VMS statement flag (&RDB&). You may also call a function or subroutine to perform the data definition with Callable RDO. In that case, use a preprocessed INVOKE DATABASE statement in the main module and the Callable RDO INVOKE DATABASE statement in the submodule.

For example, in the sample program for BASIC, the database is invoked with the GLOBAL attribute in the main program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH
```

This program calls the CALLABLE subroutine. The CALLABLE subroutine invokes the database using the RDB$INTERPRET function:

```
return_status = RDB$INTERPRET(                                  &
               'DATABASE !VAL = FILENAME "MF_PERSONNEL" ' BY DESC,&
                                          dbhandle BY DESC)
IF (return_status AND 1%) <> 1%
THEN
        CALL Callable_error_handler(return_status, retry_count,lock_error)
        success_flag = 0%
END IF
```

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the Callable RDO sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

13.5.2 Embedding Data Definition Statements Using Callable RDO

Data definition statements require a read/write transaction. When an Rdb/VMS program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You must perform Callable RDO data definition statements within a read/write transaction. However, if you start a read/write transaction in the Callable RDO portion of your program, make sure that you commit or roll back any active transactions you started in the preprocessed portion of your program first. If a transaction is active in your program when you issue the START_TRANSACTION statement with a Callable RDO statement, your Callable RDO statement will return a run-time RDO error.

If you call the RDB$INTERPRET function for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view in Callable RDO and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist, and the preprocessor generates errors for those statements that refer to either the field, relation, or view. You can define indexes and constraints and any other database elements that are not referred to in the preprocessed code.

You can perform any preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than relying on implicit START_TRANSACTION declarations.

Example 13–18, from the DDL_STMNT subroutine, shows how to perform data definition tasks in RDBPRE BASIC programs.

Example 13–18 Embedding Data Definition Statements in RDBPRE BASIC

```
Ddl_stmnt:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine demonstrates how to perform data definition tasks  !
! from an RDBPRE BASIC program.  You must use the Callable RDO       !
! interface, RDB$INTERPRET, to perform data definition tasks in      !
! preprocessed programs.                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PRINT FOR loop_cnt = 1% TO 24%
PRINT  "Execute a DDL statement "
PRINT
! Invoke the database to make it known to Callable RDO.

return_status = RDB$INTERPRET(                                  &
                        'DATABASE !VAL = FILENAME "MF_PERSONNEL" ' BY DESC,&
                                                      dbhandle BY DESC)
IF (return_status AND 1%) <> 1%
THEN
        CALL Callable_error_handler(return_status, retry_count,lock_error)
        success_flag = 0%
END IF

no_more_ddl_statements = 0%
Ddl_stmnt_1:

! Prompt user for input.  Ordinarily, it would not be likely that
! you would ask a user to define an index for the database.
! This example serves only to show you how this type of task can be
! done from within a BASIC environment.

UNTIL no_more_ddl_statements
        GOSUB Enter_ddl_statement
        EXIT ddl_stmnt_1 IF no_more_ddl_statements
        confirm = 0%

        UNTIL confirm OR no_more_ddl_statements
                PRINT "Did you enter the definition correctly (Y/N): ";
                INPUT answer
                confirm = -1% IF EDIT$(answer,32%) = "Y"
                IF NOT confirm
                THEN
                        GOSUB Enter_ddl_statement
                END IF
                IF no_more_ddl_statements
                THEN
                        return_status = RDB$INTERPRET("FINISH" BY DESC)
                        RETURN
                END IF
        NEXT
        transaction_started = 0%
        retry_count = 0%
```

(continued on next page)

Example 13–18 (Cont.) Embedding Data Definition Statements in RDBPRE BASIC

```
! Start a READ_WRITE transaction.

        UNTIL transaction_started OR retry_count > 5
                transaction_started = -1%
                return_status = RDB$INTERPRET(                  &
                                "START_TRANSACTION READ_WRITE" BY DESC)
                IF (return_status AND 1%) <> 1%
                THEN
                        CALL Callable_error_handler(return_status, &
                                                        retry_count,&
                                                        lock_error)
                        success_flag = 0%
                        transaction_started = 0%
                END IF
        NEXT
        IF transaction_started
        THEN
                success_flag = 0%
                retry_count = 0%
                lock_error = -1%
                UNTIL success_flag OR                                  &
                        (lock_error AND retry_count > 5)                &
                        OR (NOT lock_error)
                        lock_error = 0%
                        success_flag = -1%
! Pass the data definition statement specified by the user
! to RDB$INTERPRET.

                        return_status = RDB$INTERPRET(ddl_statement BY DESC)

                        IF (return_status AND 1%) <> 1%
                        THEN
                                CALL Callable_error_handler(return_status,&
                                                        retry_count,    &
                                                        lock_error)
                                success_flag = 0%
                        END IF
                NEXT
! Inform the user of the success or failure of the data definition task.

                IF success_flag
                THEN
                        PRINT "Transaction successful"
                        return_status = RDB$INTERPRET("COMMIT" BY DESC)
                ELSE
                        PRINT "Transaction failed"
                        return_status = RDB$INTERPRET("ROLLBACK" BY DESC)

                END IF
        END IF
NEXT
return_status = RDB$INTERPRET("FINISH" BY DESC)
RETURN
```

(continued on next page)

Example 13–18 (Cont.) Embedding Data Definition Statements in RDBPRE BASIC

```
Enter_ddl_statement:

! This subroutine is used to prompt user for data definition statement.

PRINT 'Please enter the data definition statement to define'
PRINT 'or delete a temporary index, or press CTRL/Z'
PRINT
PRINT 'For example, to define an index for EMPLOYEES based'
PRINT 'on EMPLOYEE_ID, you might enter: '
PRINT
PRINT 'DEFINE INDEX EMP_EMPLOYEE_ID FOR EMPLOYEES DUPLICATES ARE ALLOWED.'
PRINT 'EMPLOYEE_ID. END EMP_EMPLOYEE_ID INDEX.'
PRINT
PRINT 'To delete this index, you might enter: '
PRINT
PRINT 'DELETE INDEX EMP_EMPLOYEE_ID.'
PRINT
WHEN ERROR IN
        INPUT ddl_statement
USE
        no_more_ddl_statements = -1%
END WHEN
RETURN
```

13.6 Handling Rdb/VMS Run-Time Errors

Before reading this section, you should be familiar with the information contained in Chapter 10 of this manual. Chapter 10 discusses error handling concepts; this section contains information that, for the most part, is specific to error handling in RDBPRE BASIC.

This section describes how to detect Rdb/VMS errors that occur at run time, how to display the accompanying messages, and how to recover from errors. In most cases, this section assumes that you have debugged the program for errors in both Rdb/VMS and host language statements. This section discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your BASIC user's guide for such information.

If you choose to combine Callable RDO and RDBPRE DML statements, use separate error handling routines for each one. See Chapter 19 for information on handling Callable RDO errors.

13.6.1 Error Handling

RDBPRE BASIC enables you to detect errors with the ON ERROR clause. If an error occurs in an Rdb/VMS data manipulation statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- The ON ERROR clause
- Determining which error has occurred using the LIB$MATCH_COND run-time library routine
- Error message display using the SYS$GETMSG, SYS$PUTMSG, and LIB$SIGNAL routines

Information on creating user-supplied error messages is contained in Chapter 10.

13.6.2 Detecting Errors Using the ON ERROR Clause

You can use the ON ERROR clause only in preprocessed programs. All Rdb/VMS data manipulation statements except the INVOKE DATABASE and DECLARE_STREAM statements offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you can include one or more host language or Rdb/VMS statements, or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

Note *Do not use the BASIC REM or line number, or the START_TRANSACTION statement within the ON ERROR . . . END_ERROR block.*

If you do not use the ON ERROR clause and an Rdb/VMS error occurs, Rdb/VMS passes the error to the VMS Run-Time Library routine, LIB$STOP, which sets the severity level to 4 (FATAL) and forces program termination.

See Chapter 10 for a more complete description of the ON ERROR clause.

The following BASIC code fragment shows the placement of the ON ERROR clause and host language statements within a MODIFY operation:

```
&RDB&   FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employees::employee_id
&RDB&           MODIFY E USING
&RDB&           ON ERROR
                        success_flag = 0%
                        CALL Error_handler (RDB$STATUS,         &
                                            retry_count,    &
                                            success_flag,   &
                                            lock_error)
&RDB&           END_ERROR
&RDB&                   E.ADDRESS_DATA_1 = employees::address_data_1;
&RDB&                   E.ADDRESS_DATA_2 = employees::address_data_2;
&RDB&                   E.CITY           = employees::city;
&RDB&                   E.STATE          = employees::state;
&RDB&                   E.POSTAL_CODE    = employees::postal_code;
&RDB&           END_MODIFY
&RDB&   END_FOR
```

13.6.3 Determining Which Errors Have Occurred

After detecting an error, you want to determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation. You determine which error has occurred by evaluating the symbolic error code of the error.

13.6.3.1 Using Symbolic Error Codes All communication with Rdb/VMS is done through procedure calls. In preprocessed programs, the preprocessor converts Rdb/VMS statements to host language calls to Rdb/VMS procedures. Every procedure returns a status value into a program variable, RDB$STATUS, that is declared by the preprocessor. The return status value is a longword that identifies a unique message in the system message file. The return status value may indicate success, in which case data manipulation continues uninterrupted. Or this value may signal an error, in which case control passes to the error handler.

In RDBPRE BASIC programs, the preprocessor names this variable RDB$STATUS and declares it to be a longword. The return status value is the second element of a 20-longword array, RDB$MESSAGE_VECTOR. (The RDB$MESSAGE_VECTOR array is the message vector that Rdb/VMS uses to pass information to and from BASIC programs.)

Each error generated by an RDBPRE statement is represented as a symbolic error code. You can use these symbolic error codes to control program logic for specific errors. When the Rdb/VMS ON ERROR clause detects an error, your error handler should:

- Evaluate the symbolic error code either by calling the LIB$MATCH_COND routine or using a BASIC equality test

- Direct program logic with a BASIC host language statement such as the SELECT statement

Although symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. Chapter 10 explains why this is recommended.

13.6.3.2 Declaring Symbolic Error Codes Rdb/VMS symbolic error codes are longword values. In BASIC programs, you must declare symbolic error codes as external constants. For example:

```
! possible errors to look for:
   EXTERNAL INTEGER CONSTANT RDB$_LOCK_CONFLICT  !lock conflict
   EXTERNAL INTEGER CONSTANT RDB$_DEADLOCK       !deadlock
   EXTERNAL INTEGER CONSTANT RDB$_INTEG_FAIL     !constraint failed
```

13.6.3.3 Calling LIB$MATCH_COND When you want to determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine, LIB$MATCH_COND.

You also can evaluate the return status value directly with host language statement or statements, without calling the LIB$MATCH_COND routine. Generally, host language statements will use fewer resources than LIB$MATCH_COND. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. You must then link your program again under the new version so that the program will detect the correct error codes. The LIB$MATCH_COND routine matches only the condition ID of the return status value and is unaffected by changes in severity levels or facility names.

The LIB$MATCH_COND routine compares the first parameter to each of the remaining parameters in its parameter list. If a match is found, it returns the position in the parameter list of the matching parameter. If no match is found, the LIB$MATCH_COND routine returns a zero. You should pass the return status value to the LIB$MATCH_COND routine as the first parameter in the parameter list. In the remaining part of the parameter list, pass the error codes you wish to compare to the return status value. If one of these error codes matches the return status value, the LIB$MATCH_COND routine returns the position of the matching parameter in the parameter list.

For example, suppose you want to determine if RDB$_STREAM_EOF, RDB$_DEADLOCK, or RDB$_NOT_VALID is the return status value. Pass to the LIB$MATCH_COND routine the parameter list that contains RDB$STATUS, RDB$_STREAM_EOF, RDB$_DEADLOCK, and RDB$_NOT_VALID. If RDB$STATUS equals RDB$_DEADLOCK, then the LIB$MATCH_COND routine returns a value of 2 because RDB$_DEADLOCK is the second parameter in the parameter list.

Next, use the value that the LIB$MATCH_COND routine returns to determine the path of your error handler's conditional statement. To continue our example, assume you use a SELECT statement as the error handler's conditional statement. In this example, your SELECT statement evaluates the value returned by the LIB$MATCH_COND routine and your program falls through to the second case of the SELECT statement. Your program performs the statement or statements associated with the CASE statement. These statements might print a message to the terminal, roll back the transaction, and return program control to a point before the transaction was opened. Or they might call a more complex routine to perform these and other actions.

The BASIC format of the call to the LIB$MATCH_COND routine is:

```
err-match = LIB$MATCH_COND(ret-stat [BY REF], symb-name [BY REF]
                              [...symb-name BY REF])
```

The arguments for this BASIC call are:

- err-match

  A numeric variable that holds the integer that identifies the symbol matched.

- ret-stat

  A program variable (RDB$STATUS) that holds the return status value of the last call to the database.

- symb-name

  One or more symbolic error codes, (or the variable names you have assigned to them) that you want to match against ret-stat. The symbolic error codes are longwords and are passed by reference.

Declare the LIB$MATCH_COND routine as an external integer function.

Example 13–19 demonstrates the use of the LIB$MATCH_COND routine in a BASIC error handling routine. This error handler could be called from another program that:

- Detects errors with an ON ERROR clause
- Includes a statement within the ON ERROR . . . END_ERROR block that sets the value of a success flag to FALSE when the ON ERROR clause is executed

This error handling routine:

- Receives the return status and the success flag values
- Opens a file to record the error messages
- Uses the LIB$MATCH_COND routine to determine which error has occurred

- Uses a SELECT statement to take different actions depending on which error has occurred
- Sets the success flag to true if corrective error handling could take place
- Closes the file that records the error messages

Example 13–19 Using LIB$MATCH_COND in RDBPRE BASIC

```
SUB ERROR_HANDLER(LONG RDB$STATUS, retry_count, success_flag, lock_error_flag)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This subroutine handles run-time errors identified by      !
! the ON ERROR clause in the sample RDBPRE BASIC programs.   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

OPTION TYPE = EXPLICIT

DECLARE LONG            return_status,          &
                        seconds_to_wait

DECLARE STRING          error_record

! Declare variables, symbolic error codes, and system
! service library routines.
EXTERNAL LONG CONSTANT  RDB$_STREAM_EOF,        &
                        RDB$_DEADLOCK,          &
                        RDB$_LOCK_CONFLICT,     &
                        RDB$_INTEG_FAIL,        &
                        RDB$_NO_DUP,            &
                        RDO$_INDNOTDEF,         &
                        RDB$_NO_RECORD,         &
                        RDB$_NOT_VALID

EXTERNAL LONG FUNCTION  RDB$SIGNAL,             &
                        LIB$MATCH_COND,         &
                        LIB$SIGNAL,             &
                        LIB$CALLG,              &
                        LIB$SYS_GETMSG

COMMON (Rdb$MESSAGE_VECTOR) INTEGER     Rdb$MESSAGE_VECTOR, &
                                        Rdb$LU_STATUS, &
                                        Rdb$ALU_ARGUMENTS(17)

MAP(getmsgvars) LONG    msg_id,         &
                        msg_len,        &
                STRING  msg_txt = 132,  &
                LONG    mask,           &
                STRING  out_array = 4

seconds_to_wait = 5%
mask = 5%
```

(continued on next page)

Example 13–19 (Cont.) Using LIB$MATCH_COND in RDBPRE BASIC

```
Check_error:

! Use LIB$MATCH_COND to determine which of a series of errors
! might have occurred.

return_status  = Lib$match_cond(RDB$STATUS,             &
                                RDB$_LOCK_CONFLICT,     &
                                RDB$_DEADLOCK,          &
                                RDB$_NO_DUP,            &
                                RDB$_NOT_VALID,         &
                                RDB$_INTEG_FAIL,        &
                                RDB$_NO_RECORD)

! The CASE statement directs the program to appropriate statements
! to execute depending on the error that was detected.

SELECT return_status
        CASE 0
                GOSUB Unexpected_error
        CASE 1 to 2
                GOSUB Lock_problem
        CASE 3
                GOSUB Duplicate_not_allowed
        CASE 4
                GOSUB Invalid_data
        CASE 5
                GOSUB Integrity_failure
        CASE 6
                GOSUB Record_deleted
END SELECT
EXIT SUB

Unexpected_error:

PRINT "Unexpected error - terminating program"
OPEN "error.log" AS FILE 1%, ACCESS APPEND

return_status  = LIB$SYS_GETMSG(rdb$status BY REF,      &
                                msg_len BY REF,         &
                                msg_txt BY DESC,        &
                                mask BY REF,            &
                                out_array BY REF)
PRINT msg_txt
PRINT #1%,msg_txt
CLOSE #1%
return_status = LIB$CALLG(rdb$message_vector by ref,LOC(LIB$SIGNAL) by value)
RETURN
```

(continued on next page)

Example 13–19 (Cont.) Using LIB$MATCH_COND in RDBPRE BASIC

```
Lock_problem:

! Invoked on lock conflict or deadlock.
! Retry 5 times before rolling back.

lock_error_flag = -1%
IF (retry_count > 5)
THEN
        PRINT "Another user is accessing data you attempted to access"
        success_flag = 0%
ELSE
        SLEEP seconds_to_wait
        retry_count = retry_count + 1%
END IF
RETURN

Duplicate_not_allowed:

PRINT "You attempted to insert a record with a value already on file"
PRINT
PRINT "Please choose a new value and try again"

! Display the error message to see what index violated the
! duplicate clause.

CALL SYS$PUTMSG(RDB$MESSAGE_VECTOR)

RETURN

Invalid_data:

PRINT "In the data you entered, you specified an invalid value"
PRINT

! Display the error message to see what data was invalid

CALL SYS$PUTMSG(Rdb$MESSAGE_VECTOR)

PRINT "Please correct the error and try again"
RETURN

Integrity_failure:

PRINT "In the data you entered, you violated a constraint"
PRINT

! Display error message to see cause the integrity failure.

CALL SYS$PUTMSG(Rdb$MESSAGE_VECTOR)

PRINT "Please correct the error and try again"
RETURN

Record_deleted:

PRINT "Record entered has been deleted"
RETURN

END SUB
```

13.6.4 Displaying Error Messages

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and terminate your program, you can call the LIB$SIGNAL routine
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code

Information on creating user-supplied error messages is contained in Chapter 10.

13.6.4.1 Calling LIB$SIGNAL Call the LIB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and terminate program execution. When you call LIB$SIGNAL with LIB$CALLG, the LIB$SIGNAL routine:

- Receives the signal argument list from the signaling procedure

  This list is made up of the return status value and a set of optional arguments that provide information to condition handlers.
- Copies this signal argument list and uses it to create a signal argument vector

  The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.
- Causes a signal condition which causes the appropriate catchall condition handler to pass the signal argument vector to the SYS$PUTMSG system service

  The SYS$PUTMSG system service calls SYS$GETMSG to retrieve the message from the error messages file, and then formats and displays the error message on your terminal.
- Resignals the error

  If the error is not fatal, program execution continues. If the error is fatal, the host language error handler signals the error to the VMS default condition handler, which terminates program execution.

In BASIC, you cannot continue program execution after the call to the LIB$SIGNAL routine when the error is fatal. See Section 13.6.5 for information on how to continue program execution after a fatal error.

13.6.4.2 Methods of Calling LIB$SIGNAL The recommended method of calling LIB$SIGNAL in RDBPRE programs is to pass the message vector, RDB$MESSAGE_VECTOR, and the LIB$SIGNAL routine to the function, LIB$CALLG.

This method ensures that any FAO arguments that exist in the message vector will be formatted correctly. In addition, this method ensures that any additional error messages that clarify the nature of the program error will be returned to your program. For these reasons, Digital recommends that you always call LIB$SIGNAL with LIB$CALLG.

You can also pass the return status value, RDB$STATUS, to the LIB$SIGNAL routine. However, this method is not recommended. If you pass RDB$STATUS to the LIB$SIGNAL routine and FAO arguments exist in the Rdb/VMS error message, LIB$SIGNAL may be unable to format the Rdb/VMS error message correctly. In this case, your program may terminate abruptly or may return an incompletely formatted error message.

If your application requires that you call LIB$SIGNAL without LIB$CALLG, be certain that the error message does not contain FAO arguments. Figure 10–1 in Chapter 10 illustrates the format of the message vector.

13.6.4.3 The Format of the LIB$SIGNAL Calling Sequence with RDB$MESSAGE_VECTOR and RDB$STATUS The BASIC format of the LIB$SIGNAL calling sequence with the message vector (RDB$MESSAGE_VECTOR) is:

```
CALL LIB$CALLG(RDB$MESSAGE_VECTOR[BY REF], LIB$SIGNAL BY VALUE)
```

The LIB$SIGNAL argument is the run-time library routine that will receive RDB$MESSAGE_VECTOR. This argument is passed by reference in BASIC.

When using the LIB$CALLG routine to pass the message vector, you must declare LIB$CALLG as an external integer function in BASIC. When using this routine, you must declare LIB$SIGNAL as:

```
EXTERNAL INTEGER FUNCTION LIB$SIGNAL
```

An earlier example, Example 13–19, demonstrates how to call LIB$SIGNAL with LIB$CALLG. The BASIC format of the LIB$SIGNAL calling sequence with RDB$STATUS is:

```
CALL LIB$SIGNAL ([BY VALUE]RDB$STATUS)
```

13.6.4.4 Calling SYS$PUTMSG

Call the SYS$PUTMSG system service when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service displays the error message on the terminal and writes it to the error file designated by the logical name SYS$ERROR. You can define SYS$ERROR at the DCL level to be your program error file when you want the SYS$PUTMSG system service to write an Rdb/VMS error message to it.

The first parameter in the call to the SYS$PUTMSG system service is the message vector, RDB$MESSAGE_VECTOR. Figure 10–1 in Chapter 10 illustrates the format of the message vector. The SYS$PUTMSG system service can accept other optional parameters that specify a routine that receives control during message processing, and the facility name to be used in displaying the message (if you want the facility to be different from the default facility prefix that is associated with the message). The message vector is required; you may omit the optional parameters. See the *VMS System Services Volume* for a complete description of the SYS$PUTMSG system service.

The BASIC format of the SYS$PUTMSG calling sequence is:

```
status = SYS$PUTMSG ([BY REF] RDB$MESSAGE_VECTOR)
```

Declare the SYS$PUTMSG system service as an external integer function in BASIC. See an earlier example, Example 13–19, for a demonstration of the use of the SYS$PUTMSG system service.

13.6.4.5 Calling SYS$GETMSG

Call the SYS$GETMSG system service when you want to use an error message generated by Rdb/VMS within your program and continue program execution.

Because BASIC uses dynamic strings, you should use the VMS Run-Time Library routine, LIB$SYS_GETMSG, to call SYS$GETMSG. The LIB$SYS_GETMSG routine calls SYS$GETMSG and returns a message string using the semantics of the caller's string; in this case, a dynamic string.

The first parameter in the call to the LIB$SYS_GETMSG routine is the Rdb/VMS return status value, the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program as the second parameter of the call. You must declare a string to receive the message. Your program can then manipulate this string in any way it chooses. Your program can:

- Display the string
- Write the string to a file

You can also evaluate character substrings within the string, but Digital recommends that you do not use this method. The message text may change from one version of Rdb/VMS to the next.

The SYS$GETMSG system service requires a parameter to receive the length of the message string. You may omit the actual parameter, but you must include a comma to signify the argument. The SYS$GETMSG system service accepts other optional parameters that define what is included in the returned message and receives the FAO count of the message. You may omit these parameters; if you do, all components of the message are returned. See the *VMS System Services Volume* for further information on the SYS$GETMSG system service.

The SYS$GETMSG system service does not format the FAO arguments in the error message; instead, it returns the error message with format parameters embedded in it. If your error message contains a view name, for example, SYS$GETMSG will return the message:

```
<View !AC can not be updated>
```

You can call the SYS$FAO system service to format the FAO arguments in the message the SYS$GETMSG system service returns to your program. However, when the error message contains FAO arguments, it is preferable to call the SYS$PUTMSG system service rather than SYS$GETMSG. The optional parameters that you can specify with the LIB$SYS_GETMSG routine are not shown below. For more information on LIB$SYS_GETMSG, see the *VMS System Services Volume*.

The BASIC format of the LIB$SYS_GETMSG calling sequence is:

```
ret-stat = LIB$SYS_GETMSG(RDB$STATUS [BY REF],[msg-len BY REF],msg-string,,)
```

The arguments of this calling sequence are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- msg-len

  The number of characters written into msg-string, not counting padding in the case of a fixed-length string. The msg-len argument contains the address of a signed word integer that is this number.

  If the input string is truncated to the size specified in the msg-string descriptor, msg-len is set to this size. Therefore, msg-len can always be used by the calling program to access a valid substring of msg-string.

- msg-string

  The address of a descriptor that points to the message string. The LIB$SYS_GETMSG routine writes the message that has been returned by SYS$GETMSG into msg-string.

Declare the LIB$SYS_GETMSG routine as an external integer function. See an earlier example, Example 13–19, for a demonstration of the use of LIB$SYS_GETMSG.

13.6.5 Handling Fatal Errors

In some instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not be authorized to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error using the LIB$MATCH_COND routine or host language statement or statements to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or LIB$SYS_GETMSG routine to output an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

In other programming languages, you can also call the LIB$SIGNAL routine to display a fatal error message, but you must then use the LIB$ESTABLISH routine to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

In BASIC, the use of a condition handler is unpredictable. If you want to create your own error handler, your handler replaces the BASIC error handler. Thus, BASIC program errors are no longer handled by the host language error handler for the remainder of program execution. Instead, you must explicitly handle host language errors in your condition handler. For this reason, use of the LIB$ESTABLISH routine is not recommended in BASIC.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary before calling the LIB$SIGNAL routine. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions

- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the LIB$SIGNAL routine without establishing a condition handler, LIB$SIGNAL displays the error message and terminates your program. Perform any cleanup before making the call to LIB$SIGNAL. However, if your cleanup includes any Rdb/VMS statements (such as ROLLBACK), these new calls to the database will change the return status value contained in RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in this case, the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling the LIB$SIGNAL routine without performing any cleanup operations is not recommended.

14 Using the COBOL Program Environment

This chapter describes how to access an Rdb/VMS database using COBOL and the Rdb/VMS preprocessor interface, RDBPRE. This chapter presents the following main topics:

- Using Rdb/VMS data manipulation statements
- Using Rdb/VMS data definition statements
- Error handling in RDBPRE COBOL

Most examples in this chapter are available on line. The Rdb/VMS installation procedure writes the sample programs to SYS$COMMON:[SYSHLP.EXAMPLES.RDBVMS]. The file names for these programs are: COB_SAMPLE.RCO, COB_CALL_OTHER.RCO and COB_CALLABLE_ERROR_HANDLER.RCO. The sample program COB_SAMPLE.RCO contains most of the procedures referred to in this chapter, including an error handler for the data manipulation statements.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

Note *Before reading this chapter, you should be familiar with the information contained in Chapter 9. The main purpose of this chapter is to provide information and examples specific to VAX COBOL.*

14.1 The RDBPRE COBOL Preprocessor Interface

When you use the RDBPRE COBOL preprocessor interface, you simply include Rdb/VMS data manipulation statements directly in your program wherever you need them. You must use the special statement flag (&RDB&) with each Rdb/VMS data manipulation statement you include in your COBOL program. When you preprocess the source program, the preprocessor converts the Rdb/VMS data manipulation statements to a series of COBOL calls to Rdb/VMS. At run time, Rdb/VMS executes the calls and returns any retrieved data to the program.

Note *RDBPRE supports COBOL programs in terminal format only. Do not use ANSI format for Rdb/VMS programs.*

You cannot preprocess a program that attempts to access a non-existent database, unless your database refers to the data dictionary, CDD/Plus, and refers only to the definitions stored there. That is, if you specify a compile-time file name in the DATABASE statement, the database must exist at preprocess time. If you specify a compile-time path name in the DATABASE statement, the path name element must exist in the data dictionary at preprocess time. This is because the preprocessor must be able to validate relation and field definitions in the programs that refer to the database.

14.2 Embedding DML Statements in the RDBPRE COBOL Program Environment

The Rdb/VMS data manipulation statements are a subset of the Relational Database Operator (RDO) utility statements. With the Rdb/VMS data manipulation statements, you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of the Rdb/VMS data manipulation statements.

14.2.1 Converting an RDO Prototype to the RDBPRE COBOL Program Environment

Once you have created a prototype of your queries in the interactive RDO facility, you are ready to convert these RDO statements to the COBOL program environment. See Chapter 7 for a full discussion of creating a prototype in RDO and for examples. Example 14–1 is a COBOL program based on the RDO prototype examples in Chapter 7.

Example 14–1 Converting an RDO Prototype to RDBPRE COBOL

```
Store_cand.

****************************************************************************
* This procedure stores a record in the CANDIDATES relation.  It shows how *
* to store a value in a field of data type VARYING STRING.                 *
****************************************************************************

   DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
   DISPLAY "Store Candidates" LINE 1 COLUMN 20
   DISPLAY "" LINE 2 COLUMN 1

* Prompt the user for data to store in the CANDIDATES relation.

   DISPLAY "Please enter the first name of the candidate or type exit: " NO
   ACCEPT candidate_first_name PROTECTED REVERSED
   PERFORM UNTIL candidate_first_name = "EXIT" OR "exit"
      INITIALIZE confirm_flag
      PERFORM until confirm
          DISPLAY "Please enter the candidates middle initial: " NO ADVANCING
          ACCEPT candidate_middle_initial PROTECTED REVERSED
          DISPLAY "Please enter the last name of the candidate: "NO ADVANCING
          ACCEPT candidate_last_name PROTECTED REVERSED
          DISPLAY "Please enter candidate status information: "  NO ADVANCING
          ACCEPT  candidate_status
          DISPLAY "Have you entered the candidate
-                  " information correctly(Y/N): "               NO ADVANCING
          ACCEPT  confirm_flag PROTECTED REVERSED
      END-PERFORM

&RDB&   START_TRANSACTION READ_WRITE RESERVING CANDIDATES FOR SHARED WRITE
        MOVE 'Y' TO success_flag
* Store the values specified by the user in the CANDIDATES relation.
* Check for errors and inform the user of the success or failure of
* the STORE operation.

&RDB& STORE C IN CANDIDATES USING
&RDB&    ON ERROR
            MOVE "N" TO success_flag
            CALL "Error_handler" USING RDB$STATUS, retry_count,
                  success_flag, lock_error_flag
&RDB&    END_ERROR
&RDB&    C.LAST_NAME   = candidate_last_name;
&RDB&    C.FIRST_NAME  = candidate_first_name;
&RDB&    C.MIDDLE_INITIAL = candidate_middle_initial;
&RDB&    C.CANDIDATE_STATUS = candidate_status
&RDB& END_STORE
      IF    successful
      THEN
            DISPLAY "Update operation succeeded"
&RDB&       COMMIT
      ELSE
            DISPLAY "Update operation failed"
&RDB&       ROLLBACK
      END-IF
      DISPLAY "Please enter the first name of the candidate or type exit: " NO
      ACCEPT   candidate_first_name PROTECTED REVERSED
   END-PERFORM.
```

The syntax you use for preprocessed Rdb/VMS data manipulation statements is not identical to the statement syntax you use in RDO. When you incorporate your prototype RDO statements into a program, you need to consider these areas:

- Use of host language variables
- Use of Rdb/VMS statement flags, described in Chapter 12
- Differences in syntax
  - Using the GET statement instead of the PRINT statement
  - Nesting FETCH and GET operations within a host language loop
  - Using the ON ERROR and AT END clauses to detect error conditions
- Effects on structured programming
- Handling Rdb/VMS errors

Additionally, if you are using multiple databases and you use the COBOL line terminator, a period (.), at the end of a DATABASE statement, it should only appear in the last DATABASE statement in a series. For example:

```
&RDB& INVOKE DATABASE AA = FILENAME "MF_PERSONNEL".
&RDB& INVOKE DATABASE BB = FILENAME "SHIPPING".
```

The preceding statements will fail, but the following will succeed:

```
&RDB& INVOKE DATABASE AA = FILENAME "MF_PERSONNEL"
&RDB& INVOKE DATABASE BB = FILENAME "SHIPPING".
```

Or:

```
&RDB& INVOKE DATABASE AA = FILENAME "MF_PERSONNEL"
&RDB& INVOKE DATABASE BB = FILENAME "SHIPPING"
```

14.2.1.1 Using Host Language Variables

A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. You can use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following statements allow the use of host language variables:

- Any data manipulation statement that permits the use of an RSE
- GET
- DATABASE (you can specify a database handle)
- READY
- FINISH

When you declare host language variables, simply follow the naming rules for COBOL. Ensure that host language variable data types and sizes are compatible with the corresponding database field data types and sizes. Refer to Chapter 8 for the list of equivalent COBOL data types.

Note that you cannot use the name of a database field (a context variable and a field name) as a subscript of an array.

Example 14–2 shows the use of host language variables to store a record. The host language variables appear in lowercase.

Example 14–2 Using Host Language Variables to Store a Record in RDBPRE COBOL

```
&RDB&  STORE J IN JOBS USING
&RDB&     J.JOB_CODE       = job_code;
&RDB&     J.JOB_TITLE      = job_title;
&RDB&     J.MAXIMUM_SALARY = max_sal;
&RDB&     J.MINIMUM_SALARY = min_sal;
&RDB&     J.WAGE_CLASS     = wage_class
&RDB&  END_STORE
```

A convenient way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus. You can copy relation definitions, which include all the fields within the relation. However, you must be careful to copy only those field and relation definitions with data types that are supported by COBOL. See Chapter 12 for more information about using data dictionary definitions.

Note *You must use the COBOL keyword IN (not OF) to qualify variables used in Rdb/VMS data manipulation statements in COBOL. The preprocessor RDBPRE returns a syntax error if you use the keyword OF.*

14.2.1.2 Using Host Language Variables in Conditional Expressions You can use conditional expressions to limit the records included in a record stream. Conditional expressions contain one or more relational operators (see Table 3–1 in Section 3.5) and optionally logical operators (AND, OR, NOT).

In a programming environment, you probably do not want to code a specific value for the comparison string, as in:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING 'NH'
```

It is more likely that you want the user to supply the comparison string at run time. In this case, you need to declare a host language variable to hold the comparison string. For example:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING state_code
```

For the STARTING_WITH, MATCHING, and CONTAINING conditional expressions, you must declare your host language variable in such a way that the preprocessor can determine the correct length of the comparison string.

In COBOL, the declaration of the varying string host language variable is complex. Incorrect coding can lead to serious errors which might not be readily apparent. Because COBOL does not have a function to determine the length of a string, you must provide a numeric field to contain the length of the comparison string.

The length of the comparison string must be passed to the program either directly by the end user, or you must put code in your program that will determine the length of the comparison string. If the correct length of the comparison string is not passed to the program, two problems can result.

The first problem arises if the length of the comparison string is not passed to the program at all. In this case, the length of the comparison string will be null. As a result, all the records in the relation will be included in a record stream that is formed with the CONTAINING clause. This is because every value “contains” the null value. This can be a serious problem if you are using the CONTAINING clause to MODIFY or ERASE records. You will update every record in the relation, rather than the selected few you probably expect.

The second problem that can arise from improper COBOL coding occurs when the value passed as the length of the comparison string is too small. For example, if the value passed for the comparison string is eight or less, and you use the following RSE to form a stream of records from a database that contains the following field values, all the records will be included in the stream because the first eight characters of all the field values are the same (namely, “OPTION R”):

RSE:

```
&RDB& FOR T IN TEST_INFO WITH
&RDB&      T.FEATURE-NAME CONTAINING F-NAME IN WS-INPUT-TE AND
&RDB&      T.FEATURE-TYPE = F-TYPE
&RDB&          MODIFY T USING
&RDB&            T.TIME_EST = TIME-EST IN WS-INPUT-TE
&RDB&          END_MODIFY
&RDB& END_FOR
```

Value in field FEATURE_NAME:

```
OPTION RR
OPTION RE
OPTION RT
```

You can require the user to enter a flag to delimit the end of a string to ensure that the length of the comparison string is passed correctly. For example, Example 14–3 requires that the user terminate his or her string with a colon (:). The INSPECT verb counts the number of characters before the colon, and places that number in LEN-PART OF WS-INPUT-TE, so that Rdb/VMS has the correct length of the comparison string. Rdb/VMS assumes that the field specified in a CONTAINING expression is a varying string data type even though the field definition in the database may not be a varying string data type. The INSPECT verb then replaces the colon with a space so that the colon is not written to the database.

Example 14–3 Using a Flag to Delimit the End of a String in RDBPRE COBOL

```
*INPUT RECORD
*
* to modify time_est.
*
* 01 is record level.
01  WS-INPUT-TE.
* 03 is group level.
        03  F-NAME.
* 05 is elementary item.
            05 LEN-PART         PIC 9(4) USAGE COMP.
            05 STRING-PART      PIC X(25).
        03  TIME_EST            PIC X(2).

* used by any modify
*
01  F-TYPE                      PIC X(3).
**************************************************************
     &RDB& DATABASE FILENAME 'FT_INFO.RDB'
**************************************************************
**************************************************************
*                                                            *
*        M A I N   S U B P R O G R A M   L O G I C           *
*                                                            *
**************************************************************
PROCEDURE DIVISION GIVING STATUS-RESULT.
**************************************************************
MAIN SECTION.
BEGIN-MOD-OPT.

    SET STATUS-RESULT TO SUCCESS.
    INITIALIZE OPTION.
    INITIALIZE WS-INPUT-TE.
*
* Get info to be modified.
```

(continued on next page)

Example 14–3 (Cont.) Using a Flag to Delimit the End of a String in RDBPRE COBOL

```
        DISPLAY "ENTER CHOICE FOR MODIFICATION".
        DISPLAY "1 = MODIFY EC CLASS STATUS".
        DISPLAY "2 = MODIFY TEST STATUS".
        DISPLAY "3 = MODIFY END-DATE".
        DISPLAY "4 = MODIFY TEST WRITERS".
        DISPLAY "5 = MODIFY TIME_ESTIMATE".
        DISPLAY "6 = EXIT THIS PROGRAM".

        DISPLAY "ENTER OPTION".
        ACCEPT OPTION.

        GO TO OPT_ONE,
              OPT_TWO,
              OPT_THREE,
              OPT_FOUR,
              OPT_FIVE,
              OPT_SIX,
        DEPENDING ON OPTION.
OPT_FIVE.

    DISPLAY "What is the existing feature name?"
    ACCEPT STRING-PART OF WS-INPUT-TE.

* At this prompt, the user must enter a string in the form NAME:, for
* instance SHOW POOL:.  The colon is used to delimit the comparison
* string.  You might choose to code this differently; if you are using
* forms, you can code it to include the colon in the form so the user
* does not have to enter it.  Or, you can code it as a constant in
* working storage in such a way that it will be appended to the field
* with the UNSTRING verb.
*
* The following INSPECT verb counts the number of characters before
* the colon (:), and places that number in LEN-PART OF WS-INPUT-TE,
* so that Rdb/VMS will have the correct length of the comparison
* string.  The INSPECT verb then replaces the colon with a space so
* that the colon is not written to the database.

    INSPECT STRING-PART OF WS-INPUT-TE TALLYING
    LEN-PART OF WS-INPUT-TE FOR CHARACTERS BEFORE ":"
    REPLACING ALL ":" BY " ".

    DISPLAY "TIME ESTIMATE (2 DIGIT NUMBER)".
    ACCEPT TIME_EST OF WS-INPUT-TE.

    DISPLAY "ENTER FEATURE-TYPE, RT, SYN OR BO".
    ACCEPT F-TYPE.

    GO TO MODIFY-TE.

MODIFY-TE.

      &RDB& START_TRANSACTION READ_WRITE RESERVING
      &RDB& TEST-INFO FOR SHARED WRITE

      &RDB& FOR T IN TEST-INFO WITH
      &RDB&     T.FEATURE-NAME CONTAINING F-NAME IN WS-INPUT-TE AND
```

(continued on next page)

Example 14–3 (Cont.) Using a Flag to Delimit the End of a String in RDBPRE COBOL

```
* The RSE must state the COBOL Group Level
* name to ensure that all of F-NAME is passed.  However, both LEN-PART
* and STRING-PART of F-NAME are part of the CONTAINING clause.  First
* the length (LEN-PART) is passed, then the string (STRING-PART). If the
* user does not pass the character count (by ending the comparison string
* with a colon), LEN-PART will be zero.
* Rdb/VMS treats this as null and because all records contain
* null, all records are modified.

      &RDB&     T.FEATURE-TYPE = F-TYPE
      &RDB&     MODIFY T USING
      &RDB&              T.TIME_EST = TIME_EST IN WS-INPUT-TE
      &RDB&     END_MODIFY
      &RDB& END_FOR

      &RDB& COMMIT

      GO TO 900-EXIT.

****************************************************************
*                                                              *
*                          THE                                 *
*                          END                                 *
*                                                              *
****************************************************************
900-EXIT.
* Return to USERMAIN.

    EXIT PROGRAM.
```

14.2.1.3 Converting DATE Data Types to TEXT

DATE data types are stored in Rdb/VMS databases in encoded binary format. To display a date, your program must first retrieve the binary value and convert it to an ASCII string. This is done by using the VMS system service routine, SYS$ASCTIM, to perform the conversion.

See the *VMS System Services Volume* for more information on using SYS$ASCTIM.

Note that RDBPRE uses the run-time library routine LIB$MOVC3 to move the value from the DATE data type to the host language variable. The preprocessor declares LIB$MOVC3 as external for you; do not declare it again in your program or you may receive a fatal compile-time error.

Example 14–4 is a code fragment from the ADD_EMPLOYEES procedure that demonstrates how to display a date.

Example 14–4 Using SYS$ASCTIM System Service Routine in RDBPRE COBOL

```
WORKING-STORAGE SECTION.
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH
                      .
                      .
                      .
01    ascii_date                 PIC X(23).
01    size_of_ascii_date         PIC S9(4)        COMP VALUE 23.
                      .
                      .
                      .
&RDB&               FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = database_key(i)
&RDB&                   ON ERROR
                          MOVE 'N' TO success_flag
                          CALL "Error_handler" USING RDB$STATUS,
                                retry_count, success_flag, lock_error_flag
&RDB&                   END_ERROR
&RDB&                   GET
&RDB&                       ON ERROR
                              MOVE 'N' TO success_flag
&RDB&                       END_ERROR
&RDB&                       employee_id = E.EMPLOYEE_ID;
&RDB&                       last_name   = E.LAST_NAME;
&RDB&                       first_name  = E.FIRST_NAME;
&RDB&                       middle_initial = E.MIDDLE_INITIAL;
&RDB&                       address_data_1 = E.ADDRESS_DATA_1;
&RDB&                       address_data_2 = E.ADDRESS_DATA_2;
&RDB&                       city           = E.CITY;
&RDB&                       state          = E.STATE;
&RDB&                       postal_code    = E.POSTAL_CODE;
&RDB&                       birthday       = E.BIRTHDAY
&RDB&                   END_GET
&RDB&               END_FOR

* If the field values were successfully retrieved, then
* convert the date field from binary to a printable (ASCII) format.
* The first and last arguments to the call to SYS$ASCTIM are not
* required arguments.

                    IF successful THEN PERFORM Display_employee END-IF
                    MOVE 'Y' TO success_flag
                      .
                      .
                      .
Display_employee.
    DISPLAY SPACE
    DISPLAY "Employee id: "   employee_id
    DISPLAY "Last name:   "   last_name
    DISPLAY "First name:  "   first_name
    DISPLAY "Middle init: "   middle_initial
    DISPLAY "Address:     "   address_data_1, SPACE address_data_2
    DISPLAY "City:        "   city
    DISPLAY "State:       "   state
    DISPLAY "Postal code: "   postal_code
```

(continued on next page)

Example 14–4 (Cont.) Using SYS$ASCTIM System Service Routine in RDBPRE COBOL

```
*   Convert binary date to ASCII format.

    CALL "SYS$ASCTIM"       USING
          BY REFERENCE      size_of_ascii_date
          BY DESCRIPTOR     ascii_date
          BY REFERENCE      birthday

    DISPLAY "Birthday:     " ascii_date(1:11)
    DISPLAY SPACE.
```

14.2.1.4 Converting ASCII DATE Strings to Binary Format

Use the VMS system service routine, SYS$BINTIM, to convert ASCII DATE strings into a binary representation so the DATE fields can be stored in the database.

See the *VMS System Services Volume* for more information on using SYS$BINTIM.

Example 14–5 is a code fragment from the ADD_EMPLOYEES procedure that demonstrates how to use SYS$BINTIM in an RDBPRE COBOL program.

Example 14–5 Using SYS$BINTIM System Service Routine in RDBPRE COBOL

```
            PERFORM UNTIL valid_date
                DISPLAY "Please enter the Employee’s birthday (dd-MMM-yyyy):"
                         WITH NO ADVANCING
                ACCEPT ascii_date PROTECTED REVERSED

* Use SYS$BINTIM to convert ASCII input to binary format.

                CALL   "SYS$BINTIM"
                    USING BY DESCRIPTOR ascii_date
                          BY REFERENCE  birthday
                    GIVING              return_status
                IF   return_status IS FAILURE
                THEN DISPLAY "Invalid date format"
                ELSE MOVE "YES" TO valid_date_flag
                END-IF
            END-PERFORM
```

14.2.2 Using Literals

Use literal values to replace variables in the same way you would in any COBOL program. Literal values can be either numeric or character strings. String literals must be quoted in either double (" ") or single (′ ′) quotation marks in COBOL. You may use a literal in any Rdb/VMS data manipulation statement that accepts a host language variable.

```
&RDB&  FOR D IN DEPARTMENTS WITH
&RDB&    D.DEPARTMENT_CODE = ′ADMN′
&RDB&    GET
&RDB&      DEP_NAME = D.DEPARTMENT_NAME
&RDB&    END_GET
&RDB&  END_FOR
```

14.2.3 Forming Record Streams

In COBOL, and any language that you use to access an Rdb/VMS database, you select the records you are interested in manipulating by gathering these records into a stream. You create this stream using the Rdb/VMS data manipulation statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

14.2.4 Retrieving Records

Rdb/VMS provides you with three statements to retrieve records:

- FOR
- Two START_STREAM statements:
  - Declared START_STREAM
  - Undeclared START_STREAM

The following sections provide COBOL examples of how to form record streams and retrieve records using the FOR and START_STREAM statements.

14.2.4.1 Using the FOR Statement to Retrieve Records The FOR statement forms a record stream and provides automatic iteration for any Rdb/VMS and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

You should avoid using the COBOL line terminator period (.) within an RDBPRE FOR loop or at the end of an AT END . . . END_FETCH or ON ERROR . . . END_ERROR statement block.

Furthermore, the COBOL preprocessor translates an Rdb/VMS FOR loop into an inline COBOL PERFORM loop. You can use an ON ERROR clause that contains a GO TO statement to transfer program control out of this PERFORM loop when an error occurs in the execution of the FOR statement. However, if you then use the /CHECK=PERFORM compiler qualifier, the system generates a run-time error and the program aborts. Do not use the /CHECK qualifier if your program uses a GO TO statement to transfer control out of an Rdb/VMS FOR loop.

Example 14–6 shows a FOR statement from the DISPLAY_CAND procedure. It uses the flag "found_candidate_flag" to determine if the RSE has been satisfied. If a candidate record is found whose fields match the values in the host language variables, the success flag is set to true. If no record matches the values in the host language variables, then the success flag remains set to false.

Example 14–6 Using the FOR Statement in RDBPRE COBOL

```
&RDB&      START_TRANSACTION READ_ONLY
           INITIALIZE found_candidate_flag
&RDB&      FOR C IN CANDIDATES WITH C.FIRST_NAME = candidate_first_name
&RDB&                      AND  C.MIDDLE_INITIAL = candidate_middle_initial
&RDB&                      AND  C.LAST_NAME      = candidate_last_name

* Retrieve and display the VARYING STRING field if a record exists
* for the specified candidate.  If no record exists for this person,
* inform the user.

&RDB&          GET candidate_status = C.CANDIDATE_STATUS END_GET
               MOVE "Y" TO found_candidate_flag
               DISPLAY candidate_first_name SPACE candidate_middle_initial
                       SPACE candidate_last_name "has the following status:"
               DISPLAY SPACE
               DISPLAY candidate_status
&RDB&       END_FOR
&RDB&       COMMIT
            IF    NOT found_candidate
            THEN  DISPLAY "No such candidate on file"
            END-IF

            DISPLAY "Please enter the first name of the candidate
-                   " or type exit: " NO ADVANCING
            ACCEPT candidate_first_name PROTECTED REVERSED
         END-IF
   END-PERFORM.
```

You can include host language statements within the FOR . . . END_FOR block to process the records within the stream. However, there is an important exception to the type of statement you can include. Do not transfer control out of the FOR . . . END_FOR block unless you do not want to return. It is impossible to enter the loop again once you have exited.

You may call a module from within a FOR loop because these subroutines execute within the FOR loop context. However, you cannot use a context variable defined in the FOR block in any subroutine that is preprocessed outside the FOR block.

14.2.4.2 Using Declared Streams to Retrieve Records Rdb/VMS supports two forms of the START_STREAM statement. The *declared* START_STREAM statement and the *undeclared* START_STREAM statement. Declared streams provide all the features of the undeclared streams and more. Most importantly, undeclared streams require that the statements you use to manipulate the stream be enclosed by the START_STREAM and END_STREAM statements in your source program. Declared streams do not impose this restriction. The statements you use to manipulate the stream may appear in any order within your program as long as the DECLARE_STREAM statement appears first and the statements execute in a logical order (START_STREAM, FETCH, GET, END_STREAM).

Digital recommends that all new applications use the declared START_ STREAM statement. For this reason, only the declared START_STREAM statement is discussed in this section. Complete details on the differences between declared and undeclared START_STREAM statements are provided in Chapter 9.

Note *If you use the AT END clause in a FETCH statement, you must use the END_ FETCH clause to terminate the FETCH statement. Do not use the COBOL statement terminator within the AT END clause. This COBOL statement terminator inadvertently terminates code generated by RDBPRE in the AT END clause.*

Example 14–7, from the PAIR procedure, shows the use of the declared START_STREAM and FETCH statements. The example pairs a CANDIDATES record with an EMPLOYEES record at random. This could not be achieved with a FOR statement. You could not conditionally end a FOR loop when all the CANDIDATES records have been paired with EMPLOYEES records. A START_STREAM statement lets you do this.

Example 14–7 Using the Declared START_STREAM and FETCH Statements in RDBPRE COBOL

```
* Declare two streams: one for the CANDIDATES relation and the other
* for the EMPLOYEES relation.

&RDB& DECLARE_STREAM cands USING CA IN CANDIDATES SORTED BY CA.LAST_NAME
&RDB& DECLARE_STREAM emps  USING EM IN EMPLOYEES  SORTED BY EM.FIRST_NAME

Pair.

*********************************************************************
* This procedure demonstrates the use of the declared START_STREAM  *
* statement.  The output of this procedure is merely a random       *
* matching of each CANDIDATES record with an EMPLOYEES record.      *
*********************************************************************


&RDB& START_TRANSACTION READ_ONLY

* Open both streams and set a flag for the end-of-stream condition
* to false.

      PERFORM Open_candidates
      PERFORM Open_employees
      INITIALIZE end_of_emps_flag, end_of_cands_flag

* Fetch a record from the CANDIDATES and EMPLOYEES relations.

      PERFORM Read_a_candidate
      PERFORM Read_an_employee

* Print the employees and candidates names until the end-of-stream
* condition is met for the stream of CANDIDATES records.

      PERFORM UNTIL end_of_cands
        DISPLAY last_name, first_name, '                    ',
                candidate_last_name, candidate_first_name
        PERFORM Read_a_candidate
        IF   NOT end_of_emps
        THEN PERFORM Read_an_employee
        END-IF
      END-PERFORM

* Close both streams.

      PERFORM Close_employees
      PERFORM Close_candidates

&RDB& COMMIT.
      DISPLAY "Press any key to continue " NO ADVANCING
      ACCEPT continue_key.
```

(continued on next page)

Example 14–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE COBOL

```
* Set of procedures to control streams.  Note that the statements
* do not appear in the order that they will be executed.  This is
* a feature that declared streams have and undeclared streams do
* not have.

Open_candidates.

* Open the CANDIDATES stream.

&RDB& START_STREAM cands.

Open_employees.

* Open the EMPLOYEES stream.

&RDB& START_STREAM emps.

Read_a_candidate.

* Fetch a CANDIDATES record.

&RDB& FETCH cands
&RDB&    AT END
            MOVE 'Y' TO end_of_cands_flag
&RDB& END_FETCH
      IF  NOT end_of_cands
      THEN
&RDB&     GET
&RDB&        candidate_last_name = CA.LAST_NAME;
&RDB&        candidate_first_name= CA.FIRST_NAME;
&RDB&        candidate_status    = CA.CANDIDATE_STATUS
&RDB&     END_GET.

Read_an_employee.

* Fetch an EMPLOYEES record.

&RDB& FETCH emps
&RDB&    AT END
            MOVE 'Y' TO end_of_emps_flag
&RDB& END_FETCH
      IF   NOT end_of_emps
      THEN
&RDB&      GET
&RDB&          last_name = EM.LAST_NAME;
&RDB&          first_name = EM.FIRST_NAME;
&RDB&          employee_id = EM.EMPLOYEE_ID
&RDB&      END_GET.
```

(continued on next page)

Example 14–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE COBOL

```
Close_employees.

* Close the EMPLOYEES stream.

&RDB& END_STREAM emps.

Close_candidates.

* Close the CANDIDATES stream.

&RDB& END_STREAM cands.
```

14.2.5 Retrieving Segmented Strings

Retrieving segmented strings is a two-step process. First, you must retrieve the record that contains the segmented string field; then, you must retrieve the individual segments that comprise the segmented string field.

You may find it easier to picture a segmented string by referring to Figure 8–1 in Chapter 8.

Rdb/VMS provides you with two statements to retrieve segmented string fields:

- FOR
- START_SEGMENTED_STRING

14.2.5.1 Using the FOR Statement to Retrieve Segmented Strings You must use two streams when processing segmented string streams. Use the first FOR or START_STREAM statement to form an outer stream of records, and then use the second FOR statement to form an inner stream of segments. This inner stream formed by the second RSE identifies the segments contained in the field specified by the outer stream formed by the first RSE. Use different context variables in the inner and outer streams.

Remember that to retrieve the segmented string, you must begin at the first segment and retrieve segments in the order that they are stored, that is, sequentially.

Example 14–8 from the DISPLAY_RESUME procedure:

- Uses a FOR statement to search the database for a record with a value for the EMPLOYEE_ID field that matches the host language variable, employee_id
- Uses a second FOR statement to loop through the segments of the segmented string field for the EMPLOYEES record

- Uses the GET statement to retrieve the individual segments that comprise a segmented string
- Displays these values on the terminal

Example 14–8 Using the FOR Statement with Segmented Strings in RDBPRE COBOL

```
Display_resume.

************************************************************
* This procedure demonstrates how to retrieve a field of  *
* data type SEGMENTED STRING.                              *
************************************************************

    DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
    DISPLAY "Display Resume " LINE 1 COLUMN 5
    DISPLAY "" LINE 2 COLUMN 1

* Prompt use to enter the ID of the employee
* resume that he or she wants to view.  If user
* enters ’exit’ then exit the procedure.

    DISPLAY "Please enter the ID of the employee whose resume "
    DISPLAY "you want to display or type exit: " NO ADVANCING
    ACCEPT employee_id PROTECTED REVERSED
    PERFORM UNTIL employee_id = "EXIT" OR "exit"
&RDB&    START_TRANSACTION READ_ONLY
         INITIALIZE found_employee_flag
* Start an outer FOR loop to retrieve the employees record(s)
* with the specified ID.

&RDB&    FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id
             MOVE ’Y’ TO found_employee_flag

* Start an inner FOR loop to retrieve the segments
* of the segmented string that comprise the employee’s
* resume.

&RDB&        FOR RR IN R.RESUME
&RDB&            GET
&RDB&               resume_segment = RR.RDB$VALUE;
&RDB&               segment_length = RR.RDB$LENGTH
&RDB&            END_GET

* Display each segment as it is retrieved from the database.

                 DISPLAY resume_segment(1:segment_length)
&RDB&        END_FOR
&RDB&    END_FOR
&RDB&    COMMIT
```

(continued on next page)

Example 14–8 (Cont.) Using the FOR Statement with Segmented Strings in RDBPRE COBOL

```
* If a record with the specified ID was not found then inform
* the user.
         IF   NOT found_employee
         THEN DISPLAY 'Employee: ', employee_id, ' has no resume on file'
         END-IF
         DISPLAY SPACE

         DISPLAY "Please enter the ID of the employee whose resume "
         DISPLAY "you want to display or type exit: " NO ADVANCING
         ACCEPT employee_id PROTECTED REVERSED
    END-PERFORM.
```

The GET statement fetches only as much of the stored segment as the host language variable that receives the segment can hold. The next GET fetches the next piece of the segment. Suppose the segmented string segment size in the previous example was declared as 80 characters and the actual length of the stored segment was 100 characters. The first GET statement would fetch 80 characters of the first segment and the next GET statement would fetch the remaining 20 characters. The third GET statement would fetch 80 characters of the second segment, the next GET statement would fetch the remaining 20, and so on.

14.2.5.2 Using the START_SEGMENTED_STRING Statement to Retrieve Segmented Strings When you want to maintain program control of loop iteration, use the START_SEGMENTED_STRING statement with a record stream formed by a FOR or START_STREAM statement. You must start two streams when processing segmented string streams with the START_SEGMENTED_STRING statement.

Form an outer stream of records with a FOR or START_STREAM statement, then use the START_SEGMENTED_STRING statement to form an inner stream of segments. This inner stream identifies the segment stream that is contained in the field specified by the FOR or START_STREAM statement. When you name the segment stream, use a different name from the one you used for the outer stream name. Also, use different context variables for the outer stream and the inner segmented string stream.

The program shown in Example 14–9:

- Uses an undeclared START_STREAM statement to find all the records in the RESUMES relation with an employee ID of 12345.
- Uses a START_SEGMENTED_STRING statement to retrieve the resume of each EMPLOYEES record found by the first stream.
- Uses the GET statement to retrieve the segments that comprise the segmented string.

- Checks the return status value of the GET statement after each segment is retrieved to make sure the end-of-segmented-string condition has not been met. If this condition has not been met, the value of the current segment is printed.
- Stops processing the segmented string field when the preceding condition is met.
- Fetches the next employee record with an employee ID of 12345, if one exists.
- Closes both streams when both the START_STREAM and START_SEGMENTED_STRING end conditions have been met.
- Commits the transaction.

Example 14–9 Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. show_resume
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

&RDB&    DATABASE pers = FILENAME 'MF_PERSONNEL'


01 end_of_stream                          PIC X.

01 resume_segment                         PIC X(80).
01 segment_length                          PIC S9(4)    COMP.
01 RDB$SIGNAL     PIC S9(9) COMP VALUE IS EXTERNAL RDB$SIGNAL.

01 RDB$_SEGSTR_EOF                    PIC S9(9) COMP
      VALUE IS EXTERNAL RDB$_SEGSTR_EOF.
PROCEDURE DIVISION.
resume_example.

&RDB&    START_TRANSACTION READ_ONLY
* Find all the records in the RESUMES relation
* with an employee ID of 12345.


&RDB&    START_STREAM RESSTR USING
&RDB&             R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
&RDB&                   FETCH RESSTR
&RDB&                   END_FETCH
```

(continued on next page)

Example 14–9 (Cont.) Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE COBOL

```
* Retrieve the resume of each employee found
* with the START_STREAM statement.

&RDB&       START_SEGMENTED_STRING RINFO USING STRN IN R.RESUME
             MOVE 'N' TO end_of_stream
             PERFORM UNTIL end_of_stream = 'Y'

* Retrieve the segments that comprise the segmented string
* field.

&RDB&            GET
-                   resume_segment = STRN.RDB$VALUE;
-                   segment_length = STRN.RDB$LENGTH
&RDB&            END_GET

* Check the return status of the GET statement after each
* segment is retrieved to make sure that the end-of-
* segmented-string condition has not been met.  If this
* condition has not been met, print the value of the current segment.
* Otherwise, stop processing the stream of segments.

                 IF RDB$LU_STATUS IS NOT EQUAL TO RDB$_SEGSTR_EOF
                   THEN
                     DISPLAY resume_segment(1:segment_length)
                   ELSE
                      MOVE 'Y' TO end_of_stream
                 END-IF
             END-PERFORM

* Close both streams.

&RDB&    END_SEGMENTED_STRING RINFO
&RDB&    END_STREAM RESSTR

&RDB&    COMMIT
&RDB&    FINISH
         STOP RUN.
```

14.2.6 Retrieving Field Values

Use the GET statement to retrieve one, several, or all the field values from a database record. You can also use the GET statement to retrieve statistical values from the database.

Do not use the RDBPRE concatenation operator (|) in a GET statement. Doing so causes a preprocessing error. To concatenate fields in preprocessed programs, first use the GET statement to retrieve the individual fields and store them in separate COBOL variables. Then concatenate the COBOL variables in a COBOL statement using the COBOL STRING statement.

Section 14.2.6.1 and Section 14.2.6.2 provide examples of retrieving field and record values. Section 14.2.6.3 provides an example of retrieving statistical values.

14.2.6.1 Using the GET Statement to Retrieve Field Values When you form a record stream using the FOR statement, you include the GET statement within the FOR . . . END_FOR block to retrieve field values from the record stream. When you form a record stream using the undeclared START_STREAM statement, you include the GET statement between the START_STREAM and END_STREAM statements. When you use the declared form of the START_STREAM statement, the GET statement must execute within the START_STREAM . . . END_STREAM block, however, it does not have to appear within this block in your program.

Example 14–10, from the LIST_RECORD procedure, shows the use of the FOR and GET statements in RDBPRE COBOL.

Example 14–10 Using the FOR and GET Statements in RDBPRE COBOL

```
* For each EMPLOYEES record that has a corresponding record in
* DEGREES, print the DEGREES record.

&RDB& FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
&RDB&     FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&         GET
&RDB&            first_name = E.FIRST_NAME;
&RDB&            last_name  = E.LAST_NAME;
&RDB&            degree     = D.DEGREE;
&RDB&            degree_field = D.DEGREE_FIELD
&RDB&        END_GET
             DISPLAY "Name is: ", first_name, SPACE, last_name
                      "Degree is: ", degree
                      "Degree field is: ", degree_field
&RDB&     END_FOR
            .
            .
            .
&RDB& END_FOR
```

See an earlier example, Example 14–7, for a demonstration of how to use the START_STREAM, FETCH, and GET statements.

14.2.6.2 Using the GET * Statement to Retrieve Field Values A special form of the GET statement is the GET * statement, which lets you retrieve database values at the record level rather than the field level. You can retrieve all the fields in a record with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the records of a relation, with record field names that match the database field names. You can use the COBOL COPY FROM DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) The GET * statement in the following example retrieves all of the fields from the records of the JOB_HISTORY relation and places their values in the job_history host language record structure:

```
&RDB& FOR FIRST 1 J IN JOB_HISTORY WITH
&RDB&   J.JOB_CODE = JOB_CODE IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   GET
&RDB&     job_history = J.*
&RDB&   END_GET
&RDB& END_FOR
```

14.2.6.3 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly, without processing each record in the record stream. RDBPRE may assign a data type to the result that is different from the data type of the field referred to in the expression. See Chapter 8 for information on the data type conversions performed by statistical expressions.

Example 14–11, from the STATS procedure, uses the statistical function COUNT to find the total number of records in the EMPLOYEES relation.

Example 14–11 Using the GET Statement to Retrieve a Statistical Value in RDBPRE COBOL

```
Stats.
**********************************************************************
* This procedure displays the total number of records stored in the  *
* EMPLOYEES relation.                                                *
**********************************************************************

      DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
      DISPLAY "Statistics" LINE 1 COLUMN 20
      DISPLAY "" LINE 2 COLUMN 1

&RDB& START_TRANSACTION READ_ONLY

      DISPLAY "The number of employees in the Corporation are: " NO ADVANCING

* Use the GET statement with a statistical function to calculate the
* total number of records in the EMPLOYEES relation.

&RDB& GET number_of_employees = COUNT OF E IN EMPLOYEES END_GET

* Display the value.

      DISPLAY number_of_employees
&RDB& COMMIT

      DISPLAY SPACE
      DISPLAY "Press any key to continue " NO ADVANCING
      ACCEPT continue_key.
```

14.2.7 Updating Records Using the STORE, MODIFY, and ERASE Statements

The Rdb/VMS update statements can only be used in a read/write transaction. (You may, of course, include any valid Rdb/VMS statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

If you update a record and triggered actions have been defined for the relation containing the record, the update operation (STORE, MODIFY, or ERASE) will have the specified effect on all the relations in the database that have a foreign key relationship with the record you want to update.

If a relation-specific constraint has been defined, your ability to perform update operations may depend on the presence of matching field values in other relations. For more information on relation-specific constraints, see Section 6.6.

Include the GET statement in a read/write transaction if you intend to update any of the fields returned by the GET statement.

Note *You may not use a view to update records if that view refers to more than one relation.*

14.2.7.1 Storing Records You can insert values into one or more fields in one relation using a single STORE statement. To store more than one record in a relation, include the STORE statement within a program loop.

Example 14–12, from the ADD_EMPLOYEES procedure, stores an employee record in the EMPLOYEES relation.

Example 14–12 Storing Records in RDBPRE COBOL

```
Store_cand.

*********************************************************************
* This procedure stores a record in the CANDIDATES relation.  It    *
* shows how to store a value in a field of data type VARYING STRING.*
*********************************************************************

    DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
    DISPLAY "Store Candidates" LINE 1 COLUMN 20
    DISPLAY "" LINE 2 COLUMN 1

* Prompt the user for data to store in the CANDIDATES relation.

    DISPLAY "Please enter the first name of the candidate"
    DISPLAY "or type exit: " NO
    ACCEPT candidate_first_name PROTECTED REVERSED
    PERFORM UNTIL candidate_first_name = "EXIT" OR "exit"
        INITIALIZE confirm_flag
        PERFORM until confirm
          DISPLAY "Please enter the candidates "
-                      middle initial: " NO ADVANCING
          ACCEPT candidate_middle_initial PROTECTED REVERSED
          DISPLAY "Please enter the last name of "
-                      the candidate: "NO ADVANCING
          ACCEPT candidate_last_name PROTECTED REVERSED
          DISPLAY "Please enter candidate status "
_                      information: "  NO ADVANCING
          ACCEPT  candidate_status
          DISPLAY "Have you entered the candidate
-                   " information correctly(Y/N): "  NO ADVANCING
          ACCEPT  confirm_flag PROTECTED REVERSED
        END-PERFORM
&RDB&   START_TRANSACTION READ_WRITE
&RDB&              RESERVING CANDIDATES FOR SHARED WRITE
        MOVE ’Y’ TO success_flag

* Store the values specified by the user in the CANDIDATES relation.
* Check for errors and inform the user of the success or failure of
* the STORE operation.
```

(continued on next page)

Example 14–12 (Cont.) Storing Records in RDBPRE COBOL

```
&RDB&   STORE C IN CANDIDATES USING
&RDB&      ON ERROR
              MOVE "N" TO success_flag
              CALL "Error_handler" USING RDB$STATUS, retry_count,
                    success_flag, lock_error_flag
&RDB&      END_ERROR
&RDB&      C.LAST_NAME   = candidate_last_name;
&RDB&      C.FIRST_NAME  = candidate_first_name;
&RDB&      C.MIDDLE_INITIAL = candidate_middle_initial;
&RDB&      C.CANDIDATE_STATUS = candidate_status
&RDB&   END_STORE
        IF    successful
        THEN
              DISPLAY "Update operation succeeded"
&RDB&         COMMIT
        ELSE
              DISPLAY "Update operation failed"
&RDB&         ROLLBACK
        END-IF
        DISPLAY "Please enter the first name "
-                of the candidate or type exit: " NO
        ACCEPT   candidate_first_name PROTECTED REVERSED
    END-PERFORM.
```

14.2.7.1.1 Using the STORE * Statement to Store Records A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in a record with the STORE * statement. To use the STORE * statement, you must first declare a record structure that contains all the fields in the relation, with record field names that match the database field names. You can use the COBOL COPY FROM DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to store in the record fields and store the entire record using the STORE * statement. Example 14–13 shows the use of the STORE * statement to store job_history, a host language record structure, in the JOB_HISTORY relation.

Example 14–13 Using the STORE * Statement in RDBPRE COBOL

```
&RDB& STORE J IN PERS.JOB_HISTORY USING
&RDB&   J.* = job_history
&RDB& END_STORE
```

14.2.7.1.2 Using the CREATE_SEGMENTED_STRING Statement to Store Segmented Strings

Use the CREATE_SEGMENTED_STRING statement and the STORE statement to store segmented strings in a relation. You must use two operations to store segmented strings.

Note *See Section 9.2.6.1.2 for information about defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name with an appropriate value for storing your segmented strings.*

Note *Segmented strings cannot be updated (ERASE, MODIFY, or STORE) as part of a triggered action. For more information, see the DEFINE TRIGGER statement in the* VAX Rdb/VMS RDO and RMU Reference Manual.

Example 14–14, from the MOD_RESUME procedure, demonstrates how to read and store a resume into a segmented string from a sequential file then it shows how to use the segmented string handle to modify an existing database record.

Example 14–14 Using the CREATE_SEGMENTED_STRING Statement in RDBPRE COBOL

```
Mod_resume.

*************************************************************
* This procedure demonstrates how to modify a field of data *
* type SEGMENTED_STRING.                                    *
*************************************************************

    DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
    DISPLAY "Modify a resume" LINE 1 COLUMN 5
    DISPLAY "" LINE 2 COLUMN 1

    DISPLAY "Please enter the ID of the employee or type exit: " NO
    ACCEPT employee_id PROTECTED REVERSED
    PERFORM UNTIL employee_id = "EXIT" OR "exit"
      DISPLAY "To modify a resume, you must supply a new "
      DISPLAY " resume to replace the old resume"
      DISPLAY SPACE

* Prompt the user for the file name of the resume that will replace
* the old resume.

      DISPLAY "Please enter file name of new resume: " NO ADVANCING
      ACCEPT file-name PROTECTED REVERSED
```

(continued on next page)

Example 14–14 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in RDBPRE COBOL

```
&RDB& START_TRANSACTION READ_WRITE RESERVING RESUMES FOR SHARED WRITE

* Create a new segmented string that will hold the value
* of the new resume.

&RDB& CREATE_SEGMENTED_STRING resume_handle
      OPEN INPUT resume_file
      INITIALIZE eof_flag
      MOVE SPACES TO resume_line
      READ   resume_file
         AT END MOVE "Y" TO eof_flag
      END-READ
      PERFORM UNTIL end_of_file
&RDB&    STORE R IN resume_handle
&RDB&        USING R.RDB$VALUE = resume_line END_STORE
         MOVE SPACES TO resume_line
         READ   resume_file
            AT END MOVE "Y" TO eof_flag
         END-READ
      END-PERFORM
      CLOSE resume_file
&RDB& END_SEGMENTED_STRING resume_handle

* Modify the old resume by supplying the segmented string handle
* from the CREATE_SEGMENTED_STRING statement as the object
* of the segmented string assignment statement.

&RDB& FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id
&RDB&    MODIFY R USING
&RDB&      R.RESUME      = resume_handle
&RDB&    END_MODIFY
&RDB& END_FOR
&RDB& COMMIT

      DISPLAY "Please enter the ID of the employee "
-                or type exit: " NO
      ACCEPT employee_id PROTECTED REVERSED
  END-PERFORM.
```

14.2.7.2 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of a record in one relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Example 14–15, a COBOL program segment from the MODIFY_ADDRESS procedure, modifies a record in the EMPLOYEES relation.

Example 14–15 Modifying Records in RDBPRE COBOL

```
                   .
                   .
                   .
&RDB&        START_TRANSACTION READ_WRITE
&RDB&                RESERVING EMPLOYEES FOR SHARED WRITE

* Modify the address fields for the specified EMPLOYEES record.

&RDB&        FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
&RDB&           MODIFY E USING
&RDB&              ON ERROR
                      MOVE "N" TO success_flag
                      CALL "Error_handler" USING RDB$STATUS, retry_count,
                            success_flag,  lock_error_flag
&RDB&              END_ERROR
&RDB&                  E.ADDRESS_DATA_1 = address_data_1;
&RDB&                  E.ADDRESS_DATA_2 = address_data_2;
&RDB&                  E.CITY           = city;
&RDB&                  E.STATE          = state;
&RDB&                  E.POSTAL_CODE    = postal_code;
&RDB&            END_MODIFY
&RDB&        END_FOR
* Notify the user of the success or failure of the modify operation.

             IF    successful
             THEN  DISPLAY "Update operation succeeded"
&RDB&              COMMIT
             ELSE
                   DISPLAY "Update operation failed"
&RDB&             ROLLBACK
             END-IF
```

14.2.7.2.1 Using the MODIFY * Statement to Modify Records A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. You can use the COBOL COPY FROM DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement.

Only use the MODIFY * statement if you need to modify every field value in a record. Modifying a field by replacing one value with an identical value needlessly adds overhead to your program.. For example, your program may check constraints on a field value that *you know* is valid because it is the same value that the field presently holds.

Example 14–16 replaces the field values of an employee record in the JOB_HISTORY relation with the values in the job_history host language record structure.

Example 14–16 Using the MODIFY * Statement in RDBPRE COBOL

```
&RDB& FOR J IN JOB_HISTORY WITH
&RDB&   J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   MODIFY J USING
&RDB&     J.* = job_history
&RDB&   END_MODIFY
&RDB& END_FOR
```

14.2.7.2.2 Modifying Segmented Strings To modify a segmented string, you must first create a new segmented string with the CREATE_SEGMENTED_STRING statement and then modify the existing record by replacing the logical pointer to the old segmented string with the logical pointer to the new segmented string. You accomplish this by using the segmented string handle in an assignment statement. As Chapter 8 explains in more detail, when you store a segmented string field, you do not actually store segments into a record—you store a logical pointer to the first segment in the segmented string. Thus, by creating a new segmented string and a new segmented string id associated with it, you can modify the field in a database record that "contains" a segmented string merely by replacing the old segmented string id with a new segmented string id. When you use the segmented string handle in an assignment statement, RDBPRE understands that it is the segmented string id which is to be assigned to the record.

Note *Although you use a MODIFY statement to modify segmented strings, you are not actually modifying the individual segments that comprise the segmented string field. You are actually replacing the entire segmented string with a new segmented string.*

See an earlier example, Example 14–14, for an illustration of how this is done in COBOL.

14.2.7.3 Erasing Records You can delete one, many, or all the records from a relation using a single ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase.

Example 14–17, from the DELETE_RECORD procedure, demonstrates how to ERASE records in COBOL programs.

Note *The definition of the sample personnel database includes the trigger EMPLOYEE_ID_CASCADE_DELETE, which performs an automatic deletion of records in the relations named in ERASE statements in Example 14–17 (except for RESUMES) when the record with the matching employee ID is* *deleted from the EMPLOYEES relation. Thus, you would not need to include "cascading deletion" logic in your programs if it were already included in a trigger definition.*

Example 14–17 Erasing Records in RDBPRE COBOL

```
&RDB&       START_TRANSACTION READ_WRITE RESERVING EMPLOYEES,
&RDB&             SALARY_HISTORY, JOB_HISTORY, DEPARTMENTS,
&RDB&             DEGREES, WORK_STATUS, RESUMES FOR SHARED WRITE

&RDB&       FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = db_key
&RDB&           FOR JH IN JOB_HISTORY
&RDB&                 WITH JH.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&               ERASE JH
&RDB&            END_FOR
&RDB&            FOR SH IN SALARY_HISTORY
&RDB&                 WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&               ERASE SH
&RDB&            END_FOR
&RDB&            FOR D IN degrees WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&               ERASE D
&RDB&            END_FOR
&RDB&            FOR R IN RESUMES WITH R.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&               ERASE R
&RDB&            END_FOR
&RDB&            ERASE E
                 DISPLAY "Employee id: ", employee_id, " deleted successfully"
&RDB&       END_FOR
&RDB&       COMMIT
```

14.3 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer or address that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this key as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, the scope of a dbkey ends with the COMMIT statement. That is, a dbkey is guaranteed to point to the same record for the life of the transaction in which it is retrieved.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

The following example demonstrates how to specify the scope of the dbkey in an RDBPRE COBOL program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH
```

Suggestions on how you can take advantage of the dbkey scope are contained in Section 9.2.7.

14.4 Using Structured Programming

Programs and modules that pass through the RDBPRE preprocessor do not have unlimited freedom in structure. Calls to routines, such as the COBOL PERFORM block, or calls to subprograms and subroutines, require that you pay special attention to the context from which they are called.

Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- DECLARE_STREAM
- FOR
- GET
- START_STREAM
- END_STREAM
- FETCH
- STORE
- MODIFY
- ERASE
- CREATE_SEGMENTED_STRING
- START_SEGMENTED_STRING
- END_SEGMENTED_STRING

These individual data manipulation statements each form only part of a complex call to the database. The preprocessor generates one call to the database, using more than one data manipulation statement. For example, a MODIFY statement executes within the context of a FOR or START_STREAM statement. The call to the database can only be made using both the FOR and MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. However, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

Also keep in mind that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement, and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which it is issued.

A stream declared with the DECLARE_STREAM statement lets you place the stream of manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

For more information on the declared and undeclared START_STREAM statement, see Section 9.2.3.2. Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- DATABASE
- READY
- START_TRANSACTION
- GET
- COMMIT
- ROLLBACK
- FINISH
- DECLARE_STREAM

Remember that you must issue the DECLARE_STREAM statement before you can issue a declared START_STREAM statement, and the DATABASE statement must appear in the data declaration section of your program.

Example 14–18, from the DELETE_RECORD and CALL_OTHER procedures, demonstrates structured programming in a preprocessed COBOL program. The DELETE_RECORD procedure and the CALL_OTHER procedure are separately preprocessed and compiled. They are linked with the LINK command. The DELETE_RECORD procedure passes the value of the dbkey to the CALL_OTHER procedure. This procedure finds the record associated with the dbkey and displays this record on the terminal. Although it is not necessary to program this query in two modules, it is done here to demonstrate how to pass variables between separately preprocessed modules.

Example 14–18 Using Data Manipulation Statements in Structured Programming in RDBPRE COBOL

```
*********************************************************************
*                   Procedure DELETE_RECORD:                        *
*                                                                   *
* This procedure passes the value of the dbkey and transaction      *
* handle to the CALL_OTHER procedure.  The CALL_OTHER procedure     *
* finds and displays the employee record associated with an         *
* employee_id specified in DELETE_RECORD and then return program    *
* control to the DELETE_RECORD procedure.                           *
*********************************************************************

             MOVE "Y" TO success_flag
&RDB&        START_TRANSACTION (TRANSACTION_HANDLE trans1)
&RDB&        READ_WRITE RESERVING EMPLOYEES FOR SHARED READ

* Find the employee record that the user wants to delete.  If
* an error occurs during the FOR operation, call an error handler.

             INITIALIZE found_employee_flag
&RDB&        FOR (TRANSACTION_HANDLE trans1)
&RDB&            E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
&RDB&            ON ERROR
                    MOVE "N" TO success_flag
                    CALL "Error_handler" USING RDB$STATUS, retry_count,
                         success_flag, lock_error_flag
&RDB&            END_ERROR

* Get the dbkey of the EMPLOYEES record that the user wants to delete.

&RDB&              GET
&RDB&              ON ERROR
                      MOVE "N" TO success_flag
&RDB&              END_ERROR
&RDB&            db_key = E.RDB$DB_KEY
&RDB&            END_GET
                 MOVE "Y" TO found_employee_flag
&RDB&        END_FOR
             IF    NOT found_employee
             THEN  DISPLAY "No employee with id: ", employee_id " on file"
             ELSE  IF    successful

* Pass the dbkey to an external procedure, CALL_OTHER, to print
* out the record to which the dbkey points.  Note that using
* an external procedure is neither necessary nor recommended for
* performing this task.  It is done in this example only to show
* how values are passed between routines in an RDBPRE COBOL program.

                   THEN  CALL "Call_other" USING db_key, trans1
                   END-IF
             END-IF
&RDB&        COMMIT (TRANSACTION_HANDLE trans1)
             DISPLAY SPACE
```

(continued on next page)

Example 14–18 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE COBOL

```
* Ask user for confirmation that this is the EMPLOYEES
* record he or she wants to delete.

          IF    found_employee
          THEN  DISPLAY "Is this the employee you want to delete (Y/N): " NO
                ACCEPT confirm_flag PROTECTED REVERSED
          END-IF
           .
           .
           .

IDENTIFICATION DIVISION.
PROGRAM-ID.  Call_other.

*********************************************************************
*                   Procedure CALL_OTHER:                           *
*                                                                   *
* This procedure is passed the dbkey and transaction handle         *
* from the DELETE_RECORD procedure.  With this information, the     *
* CALL_OTHER procedure can find and display the employee record     *
* associated with an employee_id specified in DELETE_RECORD and     *
* then return program control to the DELETE_RECORD procedure.       *
*********************************************************************

DATA DIVISION.
WORKING-STORAGE SECTION.

* Because the database was invoked in the main program with
* GLOBAL attributes, refer to it here as EXTERNAL.

&RDB&   DATABASE EXTERNAL pers = FILENAME "MF_PERSONNEL"
&RDB&                            DBKEY SCOPE IS FINISH
01  employees.
    02  employee_id         PIC X(5).
    02  last_name           PIC X(14).
    02  first_name          PIC X(10).
    02  middle_initial      PIC X.
    02  address_data_1      PIC X(25).
    02  address_data_2      PIC X(25).
    02  city                PIC X(20).
    02  state               PIC X(2).
    02  postal_code         PIC X(5).
    02  birthday            PIC S9(11)V9(7) COMP.
LINKAGE SECTION.
01  db_key                  PIC X(8).
01  trans_1                 PIC S9(9) COMP.
```

(continued on next page)

Example 14–18 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE COBOL

```
PROCEDURE DIVISION USING db_key, trans_1.
Begin.

* The transaction was started in the DELETE_RECORD subroutine,
* so there is no need to start a transaction here.  Use the
* transaction handle to identify this request with the transaction
* started in DELETE_RECORD.  Use the dbkey found in the DELETE_RECORD
* subroutine to locate the correct employee record.

&RDB&    FOR (TRANSACTION_HANDLE trans_1) E IN EMPLOYEES WITH
&RDB&         E.RDB$DB_KEY = db_key
&RDB&       GET
&RDB&          employee_id = E.EMPLOYEE_ID;
&RDB&          last_name   = E.LAST_NAME;
&RDB&          first_name  = E.FIRST_NAME;
&RDB&          middle_initial = E.MIDDLE_INITIAL;
&RDB&          address_data_1 = E.ADDRESS_DATA_1;
&RDB&          address_data_2 = E.ADDRESS_DATA_2;
&RDB&          city           = E.CITY;
&RDB&          state          = E.STATE;
&RDB&          postal_code    = E.POSTAL_CODE;
&RDB&          birthday       = E.BIRTHDAY
&RDB&       END_GET

* Display the EMPLOYEES record.  Use SYS$ASCTIM to convert
* the data stored in the database in binary format to
* ASCII format.

            DISPLAY SPACE
            DISPLAY "Employee id: "   employee_id
            DISPLAY "Last name:   "   last_name
            DISPLAY "First name:  "   first_name
            DISPLAY "Middle init: "   middle_initial
            DISPLAY "Address:     "   address_data_1, SPACE address_data_2
            DISPLAY "City:        "   city
            DISPLAY "State:       "   state
            DISPLAY "Postal code: "   postal_code
&RDB&    END_FOR
         EXIT PROGRAM.

* Return program control to the DELETE_RECORD subroutine.

END PROGRAM Call_other.
```

14.4.1 Using Handles in Structured Programming

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions
- Requests

Information on when and how to use request handles is supplied in Chapter 9. The following sections discuss how to declare handles in an RDBPRE COBOL program.

14.4.2 Declaring and Initializing Handles

With the exception of the database handle, declaring handles in RDBPRE COBOL is similar to declaring any other program variable. The declaration and initialization of a database handle is done simply by specifying the handle in the DATABASE statement. You do not declare a database handle in the data declaration portion of your COBOL program. RDBPRE initializes the handle for you. You should not assign a value to a database handle with an assignment statement (or any other way).

User-specified request and transaction handles must be declared in the data declaration portion of your program. In COBOL, declare user-specified request and transaction handles as PICS9(9) COMP and initialize them to zero.

If you want to release the resources associated with a request handle, you can do so by issuing a FINISH statement, or, if you do not want to detach from the database, you can release the request by issuing a call to the RDB$RELEASE_REQUEST procedure with the following statement (where req1 is a user-supplied request handle):

```
CALL "rdb$release_request"
USING rdb$message_vector,req_handle
GIVING return_stat.
IF return_stat is FAILURE
THEN CALL "SYS$PUTMSG" USING rdb$message_vector
END-IF
```

Declare the variable that holds the return status value as PIC S9(9) COMP.

14.4.3 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

14.4.4 Declaring and Initializing Distributed Transaction Identifiers

Declaring distributed transaction identifiers in RDBPRE COBOL is similar to declaring any other program variable. Distributed transaction identifiers must be declared in the data declaration portion of your COBOL program. Declare a distributed transaction identifier as two longwords and initialize it to zero. You should not assign a value to a distributed transaction identifier with an assignment statement.

14.5 Using Callable RDO

The RDBPRE preprocessor statements do not include data definition statements. If you want to perform data definition within your preprocessed program, you must use the Callable RDO program interface. For example, during a batch process, or when others are not using the database, your program may define a temporary index on a field to facilitate Rdb/VMS performance during your program execution.

You can also use Callable RDO when your program needs the ability to form dynamic queries. That is, when your program will not know what a query is until run time. Otherwise, you should use the RDBPRE preprocessor when possible for all COBOL data manipulation operations. Preprocessed Rdb/VMS statements execute significantly faster than calls using the function RDB$INTERPRET.

When using Callable RDO, your program communicates with Rdb/VMS using the RDB$INTERPRET function. You call RDB$INTERPRET to pass your data manipulation or data definition statement to Rdb/VMS. Declare RDB$INTERPRET as an external integer (longword) function. The RDB$INTERPRET function returns a status value that indicates the success or failure of the function. The return status value is a systemwide condition value that indicates either success or a unique Rdb/VMS symbolic error code. Your program declares a longword variable to hold the return status value so you can test the success or failure of the call. (Refer to Chapter 10 and Section 14.6 in this chapter for further information on handling Rdb/VMS run-time exception conditions.)

The COBOL format of the RDB$INTERPRET sequence is:

```
CALL "RDB$INTERPRET" USING BY DESCRIPTOR'rdb-statement'
          [, [BY DESCRIPTOR] host-var,...] GIVING ret-stat.
```

The arguments for the RDB$INTERPRET function are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement

  The Rdb/VMS statement you pass to Rdb/VMS. Handle rdb-statement according to your language's rules for handling string literals or string variables.

- host-var

  A host language variable you pass to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*. You must include a by-descriptor passing mechanism when your language's default passing mechanism for the host language variable data type is not by descriptor. Refer to your COBOL reference manual for the specific format of the passing mechanism.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some Rdb/VMS statements may produce unwieldy code in the call to the RDB$INTERPRET function. Instead, assign the Rdb/VMS statement string literal to a string variable. Then pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your Rdb/VMS data manipulation statements from the mechanics of using the RDB$INTERPRET function.

Callable RDO program development is explained in detail in Chapter 19.

The following section discusses the use of the INVOKE DATABASE statement and the scope of transactions in preprocessed programs that use Callable RDO.

14.5.1 Using the DATABASE Statement with Embedded Callable RDO

You must use an INVOKE DATABASE statement in your preprocessed RDBPRE program and a separate RDO INVOKE DATABASE statement in the embedded Callable RDO statements. To ensure that the preprocessor invokes the identical database for the preprocessed and the Callable RDO portions of the program, use the same database handle in each INVOKE DATABASE statement. Invoke the database:

- In the preprocessed program, using a GLOBAL or EXTERNAL database handle.
- In the Callable RDO program, by passing the database handle to the RDB$INTERPRET function.

For more information on database handles, see the section on handles in Chapter 9.

In Callable RDO, you must pass the database handle to the RDB$INTERPRET function as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both RDBPRE and Callable RDO DATABASE statements in the same program module. The preprocessor ignores any statement that is not preceded by the Rdb/VMS statement flag (&RDB&). You may also call a function or subroutine to perform data definition with Callable RDO. In that case, use a preprocessed INVOKE DATABASE statement in the main module and the Callable RDO INVOKE DATABASE statement in the submodule.

For example, in the sample program for COBOL, the database is invoked with the GLOBAL attribute in the main program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH
```

This program calls the procedure named CALLABLE. The CALLABLE procedure invokes the database using the RDB$INTERPRET function:

```
* Invoke the database in Callable RDO.  The INVOKE DATABASE
* statement issued at the beginning of the program (using RDBPRE)
* is unknown to Callable RDO.  If an error occurs when you invoke
* the database, call an error handler.

    CALL "RDB$INTERPRET" USING
            BY DESCRIPTOR ’DATABASE !VAL = FILENAME "MF_PERSONNEL" ’,
            BY DESCRIPTOR  dbhandle
            GIVING return_status
    IF   return_status IS FAILURE
    THEN CALL "Callable_error_handler" USING return_status,
               retry_count, lock_error_flag
         MOVE ’N’ TO success_flag
    END-IF
```

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the Callable RDO sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

14.5.2 Embedding Data Definition Statements Using Callable RDO

Data definition statements require a read/write transaction. When an Rdb/VMS program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You must perform Callable RDO data definition statements within a read /write transaction. However, if you start a read/write transaction in the Callable RDO portion of your program, make sure that you commit or roll back any active transactions you started in the preprocessed portion of your program first. If a transaction is active in your program when you issue the

START_TRANSACTION statement with a Callable RDO statement, your Callable RDO statement will return a run-time RDO error.

If you call the RDB$INTERPRET function for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view in Callable RDO and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist, and the preprocessor generates errors for those statements that refer to either the field, relation, or view. You can define indexes and constraints and any other database elements that are not referred to in the preprocessed code.

You can perform any preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than relying on implicit START_TRANSACTION declarations.

Example 14–19, from the DDL_STMNT procedure, shows how to perform data definition tasks in RDBPRE COBOL programs.

Example 14–19 Embedding Data Definition Statements in RDBPRE COBOL

```
Ddl_stmnt.

********************************************************************
* This procedure demonstrates how to perform data definition tasks *
* from an RDBPRE COBOL program.  You must use the Callable RDO     *
* function, RDB$INTERPRET, to perform data definition tasks in     *
* preprocessed programs.                                           *
********************************************************************

    DISPLAY SPACE LINE 1 COLUMN 1 ERASE TO END OF SCREEN
    DISPLAY "Execute a DDL statement " LINE 1 COLUMN 5
    DISPLAY "" LINE 2 COLUMN 1

* Invoke the database to make it known to Callable RDO.

    CALL "RDB$INTERPRET" USING
            BY DESCRIPTOR 'DATABASE !VAL = FILENAME "MF_PERSONNEL" ',
            BY DESCRIPTOR  dbhandle
            GIVING return_status
    IF   return_status IS FAILURE
    THEN CALL "Callable_error_handler" USING return_status,
               retry_count, lock_error_flag
         MOVE 'N' TO success_flag
    END-IF
```

(continued on next page)

Example 14–19 (Cont.) Embedding Data Definition Statements in RDBPRE COBOL

```
* Perform procedure to prompt user for input.

    PERFORM Enter_ddl_statement
    PERFORM UNTIL no_more_ddl_statements
    INITIALIZE confirm_flag
        PERFORM UNTIL confirm OR no_more_ddl_statements
            DISPLAY "Did you enter the definition "
-                            correctly (Y/N): " NO ADVANCING
            ACCEPT confirm_flag PROTECTED REVERSED
            IF     NOT confirm
            THEN   PERFORM Enter_ddl_statement
            END-IF
        END-PERFORM
        INITIALIZE transaction_started_flag, retry_count
        PERFORM UNTIL transaction_started OR retry_count > 5
            MOVE 'Y' TO transaction_started_flag

* Start a READ_WRITE transaction.

            CALL "RDB$INTERPRET"
                USING BY DESCRIPTOR "START_TRANSACTION READ_WRITE"
                GIVING  return_status
            IF   return_status IS FAILURE
            THEN CALL "Callable_error_handler" USING return_status,
                       retry_count, lock_error_flag
                 MOVE 'N' TO success_flag, transaction_started_flag
            END-IF
        END-PERFORM
        IF   transaction_started
        THEN INITIALIZE success_flag, retry_count, lock_error_flag
             PERFORM WITH TEST AFTER UNTIL successful OR
                (lock_error AND retry_count > 5) OR (NOT lock_error)
                MOVE 'Y' TO success_flag

* Pass the data definition statement specified by the user to
* RDB$INTERPRET.

                CALL "RDB$INTERPRET" USING BY DESCRIPTOR ddl_statement
                                     GIVING return_status
                IF   return_status IS FAILURE
                THEN CALL "Callable_error_handler"
                     USING return_status, retry_count, lock_error_flag
                     MOVE 'N' TO success_flag
                END-IF
             END-PERFORM
```

(continued on next page)

Example 14–19 (Cont.) Embedding Data Definition Statements in RDBPRE COBOL

```
* Inform the user of the success or failure of the data definition
* task.

             IF   successful
             THEN DISPLAY "Transaction successful"
                  CALL "RDB$INTERPRET" USING BY DESCRIPTOR "COMMIT"
                                       GIVING return_status
             ELSE DISPLAY "Transaction failed"
                  CALL "RDB$INTERPRET" USING BY DESCRIPTOR "ROLLBACK"
                                       GIVING return_status
             END-IF
        END-IF
        PERFORM Enter_ddl_statement
    END-PERFORM
    CALL "RDB$INTERPRET" USING BY DESCRIPTOR "FINISH"
                         GIVING return_status.

Enter_ddl_statement.

* Prompt user for input.  Ordinarily, it would not be likely that
* you would ask a user to define an index for the database.
* This example serves only to show you how this type of task can
* be done within a COBOL environment.

    DISPLAY 'Please enter the data definition statement to define'
    DISPLAY 'or delete a temporary index, or type "exit"'
    DISPLAY SPACE
    DISPLAY 'For example, to define an index for EMPLOYEES based'
    DISPLAY 'on EMPLOYEE_ID, you might enter: '
    DISPLAY SPACE
    DISPLAY 'define index emp_employee_id for '
-                   employees duplicates are allowed.'
            REVERSED
    DISPLAY 'employee_id. end emp_employee_id index.' REVERSED
    DISPLAY SPACE
    DISPLAY 'To delete this index, you might enter: '
    DISPLAY SPACE
    DISPLAY 'delete index emp_employee_id.' REVERSED
    DISPLAY SPACE
    ACCEPT ddl_statement REVERSED.
```

14.6 Handling Rdb/VMS Run-Time Errors

Before reading this section you should be familiar with the information contained in Chapter 10 of this manual. Chapter 10 discusses error handling concepts; this section contains information that, for the most part, is specific to error handling in RDBPRE COBOL.

This section describes how to detect Rdb/VMS errors that occur at run time, how to display the accompanying messages, and how to recover from errors. In most cases, this section assumes that you have debugged the executing program for errors in both Rdb/VMS and host language statements. This section discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your COBOL user's guide for such information.

If you choose to combine Callable RDO and RDBPRE DML statements, use separate error handling routines for each one. See Chapter 19 for information on handling Callable RDO errors.

14.6.1 Error Handling

RDBPRE COBOL enables you to detect errors with the ON ERROR clause. If an error occurs in an Rdb/VMS data manipulation statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- The ON ERROR clause
- Determining which error has occurred using the LIB$MATCH_COND run-time library routine
- Error message display using the SYS$GETMSG, SYS$PUTMSG, and LIB$SIGNAL routines

Information on creating user-supplied error messages is contained in Chapter 10.

14.6.2 Detecting Errors Using the ON ERROR Clause

You can use the ON ERROR clause only in preprocessed programs. All Rdb/VMS data manipulation statements except the INVOKE DATABASE and DECLARE_STREAM statements offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you can include one or more host language or Rdb/VMS statements, or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

Note *Do not use the COBOL statement terminator or the START_TRANSACTION statement within the ON ERROR . . . END_ERROR block.*

Note that the COBOL preprocessor translates an Rdb/VMS FOR loop into an inline COBOL PERFORM loop. You can use an ON ERROR clause that contains a GO TO statement to transfer program control out of this PERFORM loop when an error occurs in the execution of the FOR statement. However, if you then use the /CHECK=PERFORM compiler qualifier, the system generates a run-time error and the program aborts. Do not use the /CHECK qualifier if your program uses a GO TO statement to transfer control out of an Rdb/VMS FOR loop.

If you do not use the ON ERROR clause and an Rdb/VMS error occurs, Rdb/VMS passes the error to the VMS Run-Time Library routine, LIB$STOP, which sets the severity level to 4 (FATAL) and forces program termination.

See Chapter 10 for a more complete description of the ON ERROR clause.

The following COBOL code fragment shows the placement of the ON ERROR clause and host language statements within a MODIFY operation:

```
&RDB&        FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
&RDB&           MODIFY E USING
&RDB&              ON ERROR
                      MOVE "N" TO success_flag
                      CALL "Error_handler" USING RDB$STATUS, retry_count,
                            success_flag,  lock_error_flag
&RDB&              END_ERROR
&RDB&                  E.ADDRESS_DATA_1 = address_data_1;
&RDB&                  E.ADDRESS_DATA_2 = address_data_2;
&RDB&                  E.CITY           = city;
&RDB&                  E.STATE          = state;
&RDB&                  E.POSTAL_CODE    = postal_code;
&RDB&            END_MODIFY
&RDB&        END_FOR
```

14.6.3 Determining Which Errors Have Occurred

After detecting an error, you want to determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation. You determine which error has occurred by evaluating the symbolic error code of the error.

14.6.3.1 Using Symbolic Error Codes All communication with an Rdb/VMS database is done through procedure calls. In preprocessed programs, the preprocessor converts Rdb/VMS statements to host language calls to Rdb/VMS procedures. Every procedure returns a status value into a program variable, RDB$STATUS, that is declared by the preprocessor. The return status value is a longword that identifies a unique message in the system message file. The return status value may indicate success, in which case data manipulation continues uninterrupted. Or this value may signal an error, in which case control passes to the error handler.

In RDBPRE COBOL programs, the preprocessor names this variable RDB$STATUS and declares it to be a longword. The return status value is the second element of a 20-longword array, RDB$MESSAGE_VECTOR. (The RDB$MESSAGE_VECTOR array is the message vector that Rdb/VMS uses to pass information to and from COBOL programs.)

Each error generated by an RDBPRE statement is represented as a symbolic error code. You can use these symbolic error codes to control program logic for specific errors. When the Rdb/VMS ON ERROR clause detects an error, your error handler should:

- Evaluate the symbolic error code either by calling the LIB$MATCH_COND routine or using a COBOL equality test
- Direct program logic with a COBOL host language statement such as the EVALUATE statement

Although symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. Chapter 10 explains why this is recommended.

14.6.3.2 Declaring Symbolic Error Codes Rdb/VMS symbolic error codes are longword values. In COBOL programs, you can declare symbolic error codes as elementary data items. For example:

```
WORKING-STORAGE SECTION.
* Rdb/VMS Message Symbols.
01 RDB$_DEADLOCK     PIC S9(9)  COMP VALUE EXTERNAL RDB$_DEADLOCK.
01 RDB$_LOCK_CONFLICT     PIC S9(9)  COMP VALUE EXTERNAL RDB$_LOCK_CONFLICT.
```

14.6.3.3 Calling LIB$MATCH_COND When you want to determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine, LIB$MATCH_COND.

You also can evaluate the return status value directly with host language statement or statements, without calling the LIB$MATCH_COND routine. Generally, host language statements will use fewer resources than LIB$MATCH_COND. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. You must then link your program again under the new version so that the program will detect the correct error codes. The LIB$MATCH_COND routine matches only the condition ID of the return status value and is unaffected by changes in severity levels or facility names.

The LIB$MATCH_COND routine compares the first parameter to each of the remaining parameters in its parameter list. If a match is found, it returns the position in the parameter list of the matching parameter. If no match is found, the LIB$MATCH_COND routine returns a zero. You should pass the return status value to the LIB$MATCH_COND routine as the first parameter in the parameter list. In the remaining part of the parameter list, pass the error codes you wish to compare to the return status value. If one of these error codes matches the return status value, the LIB$MATCH_COND routine returns the position of the matching parameter in the parameter list.

For example, suppose you want to determine if RDB$_STREAM_EOF, RDB$_DEADLOCK, or RDB$_NOT_VALID is the return status value. Pass to the LIB$MATCH_COND routine the parameter list that contains RDB$STATUS, RDB$_STREAM_EOF, RDB$_DEADLOCK, and RDB$_NOT_VALID. If RDB$STATUS equals RDB$_DEADLOCK, then the LIB$MATCH_COND routine returns a value of 2 because RDB$_DEADLOCK is the second parameter in the parameter list.

Next, use the value that the LIB$MATCH_COND routine returns to determine the path of your error handler's conditional statement. To continue our example, assume you use an EVALUATE statement as the error handler's conditional statement. In this example, your EVALUATE statement evaluates the value returned by the LIB$MATCH_COND routine and your program falls through to the second label of the statement. Your program performs the statement or statements associated with the WHEN statement. These statements might print a message to the terminal, roll back the transaction, and return program control to a point before the transaction was opened. Or they might call a more complex routine to perform these and other actions.

The COBOL format of the call to the LIB$MATCH_COND routine is:

```
CALL "LIB$MATCH_COND" USING [BY REFERENCE] ret-stat, symb-name[...]
             GIVING err-match.
```

The arguments for this COBOL call are:

- ret-stat

  A program variable (RDB$STATUS) that holds the return status value of the last call to the database.

- symb-name

  One or more symbolic error codes, (or the variable names you have assigned to them) that you want to match against ret-stat. The symbolic error codes are longwords and are passed by reference.

- err-match

  A numeric variable that holds the integer that identifies the symbol matched.

It is not necessary to declare the LIB$MATCH_COND routine in COBOL.

Example 14–20 demonstrates the use of the LIB$MATCH_COND routine in a COBOL error handling routine. This error handler could be called from another program that:

- Detects errors with an ON ERROR clause

- Includes a statement within the ON ERROR . . . END_ERROR block that sets the value of a success flag to FALSE when the ON ERROR clause is executed

This error handling routine:

- Receives the return status and the success flag values
- Opens a file to record the error messages
- Uses the LIB$MATCH_COND routine to determine which error has occurred
- Uses an EVALUATE statement to take different actions depending on which error has occurred
- Sets the success flag to true if corrective error handling could take place
- Closes the file that records the error messages

Example 14–20 Using LIB$MATCH_COND in RDBPRE COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Error_handler.

***********************************************************
* This program handles run-time errors identified by the  *
* ON ERROR clause in preprocessed RDBPRE COBOL programs.  *
***********************************************************

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT error_file ASSIGN 'error_log'.
DATA DIVISION.
FILE SECTION.
FD  error_file.
01  error_record     PIC X(132).

* Declare variables, including symbolic error codes and system
* service and library routines.

WORKING-STORAGE SECTION.
01   LIB$SIGNAL    PIC S9(9) COMP VALUE IS EXTERNAL LIB$SIGNAL.
01    exception_codes.
    05    RDB$_LOCK_CONFLICT        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.

    05    RDB$_DEADLOCK        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_DEADLOCK.

    05    RDB$_NO_DUP        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_NO_DUP.
```

(continued on next page)

Example 14–20 (Cont.) Using LIB$MATCH_COND in RDBPRE COBOL

```
    05    RDB$_NOT_VALID        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_NOT_VALID.

    05    RDB$_INTEG_FAIL        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_INTEG_FAIL.

    05    RDB$_STREAM_EOF        PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_STREAM_EOF.

    05    RDB$_NO_RECORD         PIC S9(9) COMP
          VALUE IS EXTERNAL RDB$_NO_RECORD.
01 return_status                PIC S9(9) COMP.
01 RDB$MESSAGE_VECTOR EXTERNAL.
    03 Rdb$LU_NUM_ARGUMENTS     PIC S9(9) COMP.
    03 Rdb$LU_STATUS            PIC S9(9) COMP.
    03 Rdb$ALU_ARGUMENTS        OCCURS 18 TIMES.
        05 Rdb$LU_ARGUMENTS     PIC S9(9) COMP.
01    seconds_to_wait            COMP-1        VALUE 5.

01  getmsgvars.
    05  msg_id                  PIC 9(9) COMP.
    05  msg_len                 PIC 9(9) COMP.
    05  msg_txt                 PIC X(132).
    05  mask                    PIC 9(9)  COMP VALUE 15.
    05  out_array               PIC X(4).

LINKAGE SECTION.
01 RDB$STATUS                   PIC S9(9) COMP.
01 retry-count                  PIC S9(4) COMP.
01 success_flag                 PIC X.
01 lock_error_flag              PIC X.

PROCEDURE DIVISION USING RDB$STATUS, retry_count, success_flag, lock_error_flag.
Check_error.

* Use LIB$MATCH_COND to determine which of a series of
* errors might have occurred.

    CALL "Lib$match_cond"  USING     RDB$STATUS,
                                     RDB$_LOCK_CONFLICT
                                     RDB$_DEADLOCK,
                                     RDB$_NO_DUP,
                                     RDB$_NOT_VALID,
                                     RDB$_INTEG_FAIL
                                     RDB$_NO_RECORD
                           GIVING    return_status
```

(continued on next page)

Example 14–20 (Cont.) Using LIB$MATCH_COND in RDBPRE COBOL

```
* The COBOL EVALUATE statement directs program to appropriate
* statements to execute depending on the error that
* was identified.

    EVALUATE return_status
        WHEN 0         PERFORM Unexpected_error
        WHEN 1 THRU 2  PERFORM Lock_problem
        WHEN 3         PERFORM Duplicate_not_allowed
        WHEN 4         PERFORM Invalid_data
        WHEN 5         PERFORM Integrity_failure
        WHEN 6         PERFORM Record_deleted
    END-EVALUATE
    DISPLAY SPACE
    EXIT PROGRAM.

Unexpected_error.
    DISPLAY "Unexpected error - terminating program"
    OPEN EXTEND error_file
    CALL "SYS$GETMSG" USING BY VALUE rdb$status
                            BY REFERENCE msg_len
                            BY DESCRIPTOR msg_txt
                            BY VALUE      mask
                            BY REFERENCE  out_array
                      GIVING return_status
    MOVE msg_txt(1:msg_len) TO error_record
    DISPLAY error_record
    WRITE error_record
    CLOSE error_file
    CALL "LIB$CALLG" USING BY REFERENCE Rdb$MESSAGE_VECTOR
                           BY VALUE LIB$SIGNAL.

Lock_problem.

* Invoked on lock conflict or deadlock.
* Retry 5 times before rolling back.

    MOVE 'Y' TO lock_error_flag
    IF   (retry-count > 5)
    THEN DISPLAY "Another user is accessing data you attempted to access"
         MOVE "N" TO success_flag
    ELSE CALL "LIB$WAIT" USING seconds_to_wait
         ADD 1 TO retry-count
    END-IF.

Duplicate_not_allowed.
    DISPLAY "You attempted to insert a record with a value already on file"
    DISPLAY SPACES

*   Display the error message to see
*   what index violated duplicate clause.

    CALL "SYS$PUTMSG" USING BY REFERENCE Rdb$MESSAGE_VECTOR
    DISPLAY "Please choose a new value and try again".

Invalid_data.
    DISPLAY "In the data you entered, you specified an invalid value"
    DISPLAY SPACES.

*   Display the error message to see what data was invalid.
```

(continued on next page)

Example 14–20 (Cont.) Using LIB$MATCH_COND in RDBPRE COBOL

```
    CALL "SYS$PUTMSG" USING BY REFERENCE Rdb$MESSAGE_VECTOR
    DISPLAY "Please correct the error and try again".

Integrity_failure.
    DISPLAY "In the data you entered, you violated a constraint"
    DISPLAY SPACES.

*   Display the error message to see what constraint was violated.

    CALL "SYS$PUTMSG" USING BY REFERENCE Rdb$MESSAGE_VECTOR
    DISPLAY "Please correct the error and try again".

Record_deleted.
    DISPLAY "Record entered has been deleted".

END PROGRAM Error_handler.
```

14.6.4 Displaying Error Messages

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and terminate your program, you can call the LIB$SIGNAL routine
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code

Information on creating user-supplied error messages is contained in Chapter 10.

14.6.4.1 Calling LIB$SIGNAL Call the LIB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and terminate program execution. When you call LIB$SIGNAL with LIB$CALLG, the LIB$SIGNAL routine:

- Receives the signal argument list from the signaling procedure

  This list is made up of the return status value and a set of optional arguments that provide information to error handlers.
- Copies this signal argument list and uses it to create a signal argument vector

  The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.

- Causes a signal condition which causes the appropriate catchall condition handler to pass the signal argument vector to the SYS$PUTMSG system service

  The SYS$PUTMSG system service calls SYS$GETMSG to retrieve the message from the error messages file, and then formats and displays the error message on your terminal.

- Resignals the error

  If the error is not fatal, program execution continues. If the error is fatal, the host language error handler signals the error to the VMS default condition handler, which terminates program execution.

In COBOL, you cannot continue program execution after the call to the LIB$SIGNAL routine when the error is fatal. See the section on handling fatal errors in preprocessed COBOL programs in this chapter for information on how to continue program execution after a fatal error.

14.6.4.2 Methods of Calling LIB$SIGNAL The recommended method of calling LIB$SIGNAL in RDBPRE programs is to pass the message vector, RDB$MESSAGE_VECTOR, and the LIB$SIGNAL routine to the run-time library routine, LIB$CALLG.

This method ensures that any FAO arguments that exist in the message vector will be formatted correctly. In addition, this method ensures that any additional error messages that clarify the nature of the program error will be returned to your program. For these reasons, Digital recommends that you always call LIB$SIGNAL with LIB$CALLG.

You can also pass the return status value, RDB$STATUS, to the LIB$SIGNAL routine. However, this method is not recommended. If you pass RDB$STATUS to the LIB$SIGNAL routine and FAO arguments exist in the Rdb/VMS error message, LIB$SIGNAL may be unable to format the Rdb/VMS error message correctly. In this case, your program may terminate abruptly or may return an incompletely formatted error message.

If your application requires that you call LIB$SIGNAL without LIB$CALLG, be certain that the error message does not contain FAO arguments.
Figure 10–1 in Chapter 10 illustrates the format of the message vector.

14.6.4.3 The Format of the LIB$SIGNAL Calling Sequence with RDB$MESSAGE_VECTOR and RDB$STATUS The COBOL format of the LIB$SIGNAL calling sequence with the message vector (RDB$MESSAGE_VECTOR) is:

```
CALL "LIB$CALLG" USING [BY REFERENCE] RDB$MESSAGE_VECTOR,
                 BY VALUE LIB$SIGNAL.
```

The LIB$SIGNAL argument is the run-time library routine that will receive RDB$MESSAGE_VECTOR. This argument is passed by reference in COBOL.

It is not necessary to declare LIB$CALLG in COBOL. However, when using LIB$CALLG, you must declare LIB$SIGNAL as:

```
01 LIB$SIGNAL           PIC S9(9) COMP VALUE EXTERNAL LIB$SIGNAL.
```

An earlier example, Example 14–20, demonstrates how to call LIB$SIGNAL with LIB$CALLG. The COBOL format of the LIB$SIGNAL calling sequence with the return status value is:

```
CALL "LIB$SIGNAL" USING [BY VALUE] RDB$STATUS.
```

14.6.4.4 Calling SYS$PUTMSG Call the SYS$PUTMSG routine when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service displays the error message on the terminal and writes it to the error file designated by the logical name SYS$ERROR. You can define SYS$ERROR at the DCL level to be your program error file when you want the SYS$PUTMSG system service to write an Rdb/VMS error message to it.

The first parameter in the call to the SYS$PUTMSG system service is the message vector RDB$MESSAGE_VECTOR. Figure 10–1 in Chapter 10 illustrates the format of the signal message argument vector. The SYS$PUTMSG system service can accept other optional parameters that specify a routine that receives control during message processing, and the facility name to be used in displaying the message (if you want the facility to be different from the default facility prefix that is associated with the message). The signal message vector is required; you may omit the optional parameters. See the *VMS System Services Volume* for a complete description of the SYS$PUTMSG system service.

The COBOL format of the SYS$PUTMSG calling sequence is:

```
CALL SYS$PUTMSG" USING BY REFERENCE RDB$MESSAGE_VECTOR.
```

It is not necessary to declare SYS$PUTMSG in COBOL. See an earlier example, Example 14–20, for a demonstration of the use of the SYS$PUTMSG system service.

14.6.4.5 Calling SYS$GETMSG Call the SYS$GETMSG system service when you want to use an error message generated by Rdb/VMS within your program and continue program execution.

The first parameter in the call to the SYS$GETMSG system service is the Rdb/VMS return status value, the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program as the second parameter of the call. You must declare a string to receive the message. Your program can then manipulate this string in any way it chooses. Your program can:

- Display the string
- Write the string to a file

You can also evaluate character substrings within the string, but Digital recommends that you do not use this method. The message text may change from one version of Rdb/VMS to the next.

The SYS$GETMSG system service requires a parameter to receive the length of the message string. You may omit the actual parameter, but you must include a comma or a COBOL placemarker to signify the argument. The SYS$GETMSG system service accepts other optional parameters that define what is included in the returned message and receives the FAO count of the message. You may omit these parameters; if you do, all components of the message are returned. See the *VMS System Services Volume* for further information on the SYS$GETMSG system service.

The SYS$GETMSG system service does not format the FAO arguments in the error message; instead, it returns the error message with format parameters embedded in it. If your error message contains a view name, for example, SYS$GETMSG will return the message:

```
<View !AC can not be updated>
```

You can call the SYS$FAO system service to format the FAO arguments in the message SYS$GETMSG returns to your program. However, when the error message contains FAO arguments, it is preferable to call the SYS$PUTMSG system service rather than SYS$GETMSG. The optional parameters that you can specify with the SYS$GETMSG system service are not shown below. For more information on SYS$GETMSG, see the *VMS System Services Volume*.

The COBOL format of the SYS$GETMSG calling sequence is:

```
CALL "SYS$GETMSG" USING [BY VALUE] RDB$STATUS,
         [BY REFERENCE msg-len], [BY DESCRIPTOR]msg-string
          GIVING ret-stat.
```

The arguments of this calling sequence are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- msg-len

  A word that holds the number of characters written into msg-string. This is not an optional parameter; if you omit it, you must use a comma or one of the COBOL placemarkers, OMITTED or VALUE IS 0. This is passed by reference.

- msg-string

  A string variable that holds the returned error message. The maximum length of any message that can be returned is 256 bytes. This is passed by descriptor.

It is not necessary to declare SYS$GETMSG in COBOL. See an earlier example, Example 14–20, for a demonstration of the use of SYS$GETMSG.

14.6.5 Handling Fatal Errors

In many instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error using the LIB$MATCH_COND routine or host language statement or statements to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or SYS$GETMSG system service to output an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

In other programming languages, you can also call the LIB$SIGNAL routine to display a fatal error message, but you must use the LIB$ESTABLISH routine to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

In COBOL, the use of a condition handler is unpredictable. If you want to create your own error handler, your handler replaces the COBOL condition handler. Thus, COBOL program errors are no longer handled by the host language error handler for the remainder of program execution. Instead, you must explicitly handle host language errors in your condition handler. For this reason, use of the LIB$ESTABLISH routine is not recommended in COBOL.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary before calling the LIB$SIGNAL routine. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions
- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the LIB$SIGNAL routine without establishing a condition handler, LIB$SIGNAL displays the error message and terminates your program. Perform any cleanup before making the call to LIB$SIGNAL. However, if your cleanup includes any Rdb/VMS statements (such as ROLLBACK), these new calls to the database will change the return status value contained in RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in this case, the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling the LIB$SIGNAL routine without performing any cleanup operations is not recommended.

15 Using the FORTRAN Program Environment

This chapter describes how to access an Rdb/VMS database using FORTRAN and the Rdb/VMS FORTRAN preprocessor interface, RDBPRE. This chapter presents the following main topics:

- Using Rdb/VMS data manipulation statements
- Using Rdb/VMS data definition statements
- Error handling in RDBPRE FORTRAN

Most examples in this chapter are available on line. The Rdb/VMS installation procedure writes the sample programs to the directory identified by the logical name RDM$DEMO. The file names for these programs are F_SAMPLE.RFO, F_CALLABLE.FOR, F_CALLABLE_ERROR_HANDLER.FOR and F_DDL_STMNT.FOR. The sample program F_SAMPLE.RFO calls many other programs; these programs are listed in the first commented section of F_SAMPLE.RFO.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

Note *Before reading this chapter, you should be familiar with the information contained in Chapter 9. The main purpose of this chapter is to provide information and examples specific to VAX FORTRAN.*

15.1 The RDBPRE FORTRAN Preprocessor Interface

When you use the RDBPRE FORTRAN preprocessor interface, you simply include Rdb/VMS data manipulation statements directly in your program wherever you need them. You must use the special statement flag (&RDB&) with each Rdb/VMS data manipulation statement you include in your FORTRAN program. When you preprocess the source program, the preprocessor converts the Rdb/VMS data manipulation statements to a series of FORTRAN calls to Rdb/VMS. At run time, Rdb/VMS executes the calls and returns any requested data to the program.

You cannot preprocess a program that attempts to access a non-existent database, unless your database refers to the data dictionary, CDD/Plus, and refers only to the definitions stored there. That is, if you specify a compile-time file name in the DATABASE statement, the database must exist at preprocess time. If you specify a compile-time path name in the DATABASE statement, the path name element must exist in the data dictionary at preprocess time. This is because the preprocessor must be able to validate relation and field definitions in the programs that refer to the database.

15.2 Embedding DML Statements in the RDBPRE FORTRAN Program Environment

The Rdb/VMS data manipulation statements are a subset of the Relational Database Operator (RDO) utility statements. With the Rdb/VMS data manipulation statements you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of the Rdb/VMS data manipulation statements.

15.2.1 Converting an RDO Prototype to the RDBPRE FORTRAN Program Environment

Once you have created a prototype of your queries in the interactive RDO facility, you are ready to convert these RDO statements to the FORTRAN program environment. See Chapter 7 for a full discussion of creating a prototype in RDO and for examples.

You cannot use the FORTRAN /EXTEND_SOURCE qualifier in embedded data manipulation statements. Text must be between columns 7 and 72 inclusive. The use of tabs will change the “apparent” value of column 72. When you use the FORTRAN continuation character to continue an Rdb/VMS statement, do not mix tab and space characters in columns 1 to 7. For example, the word “this” starts in “apparent” column 7 in each of the following lines:

```
      this (starts with 6 spaces)
        this (starts with a tab)
        1this (starts with tab-number, the number 1 in the apparent
                column 6 is a continuation character)
```

Example 15–1 is a FORTRAN subroutine based on the RDO prototype examples in Chapter 7.

Example 15–1 Converting an RDO Prototype to RDBPRE FORTRAN

```
        SUBROUTINE store_cand

C-------------------------------------------------
C This subroutine stores a record in the
C CANDIDATES relation.  It shows how to store
C a value in a field of data type VARYING STRING.
C-------------------------------------------------

        IMPLICIT NONE
        LOGICAL success
        INTEGER retry_count

C----------------------------------------------------
C Declare variables to hold user input.  Declare the
C field that will hold the value for the field of
C data type VARYING STRING as a character string.
C----------------------------------------------------

        CHARACTER candidate_lname*14,candidate_name*10,candidate_mi
        CHARACTER candidate_status*256,confirm

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

        WRITE (6,90)
90      FORMAT ('1',T25,'**** STORE CANDIDATE ****'///)
C----------------------------------------------------
C Prompt user for data to store in the CANDIDATES
C relation.
C----------------------------------------------------

100     TYPE 110
110     FORMAT ('$',' Please enter the first name of the
        1candidate or type exit: ')
        ACCEPT 120, candidate_name
120     FORMAT (A)

        DO WHILE ((candidate_name.NE.'EXIT ') .AND.
        1               (candidate_name.NE.'exit '))
            confirm = 'N'
            DO WHILE (confirm .EQ. 'N')
                TYPE 1000
1000            FORMAT ('$',' Please enter the candidate middle initial: ')
                ACCEPT 1010, candidate_mi
1010            FORMAT (A)
```

(continued on next page)

Example 15–1 (Cont.) Converting an RDO Prototype to RDBPRE FORTRAN

```
                TYPE 2000
2000            FORMAT ('$',' Please enter the candidate last name: ')
                ACCEPT 2010, candidate_lname
2010            FORMAT (A)

                TYPE 3000
3000            FORMAT ('$',' Please enter the candidate status info: ')
                ACCEPT 3010, candidate_status
3010            FORMAT (A)

                PRINT *, '  '
                TYPE 10000
10000           FORMAT ('$',' Have you entered all data correctly? (Y/N): ')
                ACCEPT 10010, confirm
10010           FORMAT (A)
            END DO
                success = .TRUE.

&RDB&           START_TRANSACTION READ_WRITE RESERVING
&RDB&                   CANDIDATES FOR SHARED WRITE

C-----------------------------------------------------
C Store the values specified by the user in the
C CANDIDATES relation.  Inform user of the success
C or failure of the store operation.
C-----------------------------------------------------

&RDB&               STORE C IN CANDIDATES USING
&RDB&                   ON ERROR
                                success = .FALSE.
                                retry_count = retry_count + 1
                                CALL error_handler(RDB$STATUS,success)
                                IF (success) THEN
                                        retry_count = 5
                                END IF
&RDB&                   END_ERROR
&RDB&                   C.LAST_NAME = candidate_lname;
&RDB&                   C.FIRST_NAME = candidate_name;
&RDB&                   C.MIDDLE_INITIAL = candidate_mi;
&RDB&                   C.CANDIDATE_STATUS = candidate_status;
&RDB&               END_STORE

                IF (success) THEN
                        PRINT *, ' Update operation successful'
&RDB&                   COMMIT
                ELSE
                        PRINT *, ' Update operation failed'
&RDB&                   ROLLBACK
                END IF

                PRINT *, '  '
                TYPE 110
                ACCEPT 120, candidate_name

        END DO
```

(continued on next page)

Example 15–1 (Cont.) Converting an RDO Prototype to RDBPRE FORTRAN

```
        RETURN
        END
```

The syntax you use for preprocessed Rdb/VMS data manipulation statements is not identical to the statement syntax you use in RDO. When you incorporate your prototype RDO statements into a program, you need to consider these areas:

- Use of host language variables
- Use of Rdb/VMS statement flags, described in Chapter 12
- Differences in syntax
  - Using the GET statement instead of the PRINT statement
  - Nesting FETCH and GET operations within a host language loop
  - Using the ON ERROR and AT END clauses to detect error conditions
- Effects on structured programming
- Handling Rdb/VMS errors

15.2.1.1 Using Host Language Variables A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. You can use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following statements allow the use of host language variables:

- Any data manipulation statement that permits the use of an RSE
- GET
- DATABASE (you can specify a database handle)
- READY
- FINISH

When you declare host language variables, follow the naming rules for FORTRAN. Ensure that host language variable data types and sizes are compatible with the corresponding database field data types and sizes. Refer to Chapter 8 for the list of equivalent FORTRAN data types.

Note that you cannot use the name of a database field (a context variable and a field name) as a subscript of an array.

Example 15–2 shows the use of host language variables to store a record. The host language variables appear in lowercase.

Example 15–2 Using Host Language Variables to Store a Record in RDBPRE FORTRAN

```
&RDB&  STORE J IN JOBS USING
&RDB&     J.JOB_CODE       = job_code;
&RDB&     J.JOB_TITLE      = job_title;
&RDB&     J.MAXIMUM_SALARY = max_sal;
&RDB&     J.MINIMUM_SALARY = min_sal;
&RDB&     J.WAGE_CLASS     = wage_class;
&RDB&  END_STORE
```

A convenient way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus. You can copy relation definitions, which include all the fields within the relation. However, you must be careful to copy only those field and relation definitions with data types that are supported by FORTRAN. See Chapter 12 for more information about using data dictionary definitions.

15.2.1.2 Using Host Language Variables in Conditional Expressions

You can use conditional expressions to limit the records included in a record stream. Conditional expressions contain one or more relational operators (see Table 3–1 in Section 3.5) and optionally logical operators (AND, OR, NOT).

In a programming environment, you probably do not want to code a specific value for the comparison string, as in:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING 'NH'
```

It is more likely that you want the user to supply the comparison string at run time. In this case, you need to declare a host language variable to hold the comparison string. For example:

```
FOR E IN EMPLOYEES WITH E.STATE MATCHING state_code
```

For the STARTING_WITH, MATCHING, and CONTAINING conditional expressions, you must declare your host language variable in such a way that the preprocessor can determine the correct length of the comparison string.

In FORTRAN, declare the host language variable as a string, and then use the substring feature (stat_code(1:2), for example) in your RSE. The substring will permit the preprocessor, using the FORTRAN function LEN, to determine the length of the string. For example:

```
IMPLICIT NONE
CHARACTER*15 name, city
CHARACTER*4 state_code
C------------------------------------------------------
C       Program statements
C       Rdb/VMS statements:
C         invoke database, start_transaction,
C         and so on
C------------------------------------------------------
             .
             .
             .
&RDB&  FOR E IN EMPLOYEES WITH
&RDB&      E.STATE MATCHING state_code(1:2)
&RDB&   GET
&RDB&       name = E.LAST_NAME;
&RDB&       city = E.CITY;
&RDB&   END_GET
```

15.2.1.3 Converting DATE Data Types to TEXT DATE data types are stored in Rdb/VMS databases in encoded binary format. To display a date, your program must first retrieve the binary value and convert it to an ASCII string. This is done by using the VMS system service routine, SYS$ASCTIM, to perform the conversion.

Note that RDBPRE uses the run-time library routine LIB$MOVC3 to move the value from the DATE data type to the host language variable. The preprocessor declares LIB$MOVC3 as external for you; do not declare it again in your program or you may receive a fatal compile-time error.

See the *VMS System Services Volume* for more information on using SYS$ASCTIM.

Example 15–3 is a code fragment from the F_ADD_EMPLOYEES.RFO subroutine that demonstrates how to display a date.

Example 15–3 Using SYS$ASCTIM System Service Routine in RDBPRE FORTRAN

```
        SUBROUTINE add_employees

C-------------------------------------------------
C This subroutine adds a new employee to the
C EMPLOYEES relation.
C-------------------------------------------------

        IMPLICIT NONE
        LOGICAL valid_date,success
        INTEGER retry_count,number_employees_added,i
        INTEGER*4 STATUS,birthday(2),SYS$BINTIM,SYS$ASCTIM
        CHARACTER employee_id*5,last_name*14,first_name*10,middle_initial
        CHARACTER city*20,state*2,postal_code*5,ascii_date*23,confirm
        CHARACTER*25 address_data_1,address_data_2
        CHARACTER see_all
        CHARACTER*8 data_base_key,database_key(20)
&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH
                            .
                            .
                            .
&RDB&                   FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = database_key(i)
                            .
                            .
                            .
&RDB&                   GET
&RDB&                           ON ERROR
                                        success = .FALSE.
&RDB&                           END_ERROR
&RDB&                           employee_id = E.EMPLOYEE_ID;
&RDB&                           last_name = E.LAST_NAME;
&RDB&                           first_name = E.FIRST_NAME;
&RDB&                           middle_initial = E.MIDDLE_INITIAL;
&RDB&                           address_data_1 = E.ADDRESS_DATA_1;
&RDB&                           address_data_2 = E.ADDRESS_DATA_2;
&RDB&                           city = E.CITY;
&RDB&                           state = E.STATE;
&RDB&                           postal_code = E.POSTAL_CODE;
&RDB&                           birthday = E.BIRTHDAY;
&RDB&                   END_GET
&RDB&                   END_FOR
                        IF (success) THEN
C-----------------------------------------------------------------
C If the field values were successfully retrieved,
C then convert the date field from binary to ASCII format.
C The first and last arguments to the call to SYS$ASCTIM are not
C required arguments.
C----------------------------------------------------------------
```

(continued on next page)

Example 15–3 (Cont.) Using SYS$ASCTIM System Service Routine in RDBPRE FORTRAN

```
                        CALL SYS$ASCTIM (, ascii_date, birthday, )
                        TYPE 12000, employee_id, last_name, first_name,
        1                       middle_initial, address_data_1,
        1                       address_data_2, city, state,
        1                       postal_code, ascii_date
12000                   FORMAT (//   ' Employee_id:    ',A/
        1                       ' Last name:      ',A/
        1                       ' First name:     ',A/
        1                       ' Middle initial: ',A/
        1                       ' Address:        ',A,' ',A/
        1                       ' City:         ',A/
        1                       ' State:          ',A/
        1                       ' Postal code:    ',A/
        1                       ' Birthday:       ',A//)
                        success = .TRUE.
                        END IF
```

15.2.1.4 Converting ASCII DATE Strings to Binary Format Use the VMS system service routine, SYS$BINTIM, to convert ASCII DATE strings into a binary representation so the DATE fields can be stored in the database.

See the *VMS System Services Volume* for more information on using SYS$BINTIM.

Example 15–4 is a code fragment from the F_ADD_EMPLOYEES.RFO subroutine that demonstrates how to use SYS$BINTIM in an RDBPRE FORTRAN program.

Example 15–4 Using SYS$BINTIM System Service Routine in RDBPRE FORTRAN

```
     .
     .
     .
C---------------------------------------------------
C Prompt user to input date, keep prompting
C until user enters date in proper format.
C---------------------------------------------------

                DO WHILE (.NOT.(valid_date))
                        TYPE 4000
4000                    FORMAT ('$',' Please enter the Employees
        1birthday (dd-MMM-yyyy): ')
                        ACCEPT 4010, ascii_date
4010                    FORMAT (A)

C---------------------------------------------------
C Use SYS$BINTIM to convert ASCII input to binary
C format.
C---------------------------------------------------

                        STATUS = SYS$BINTIM (ascii_date, birthday)
                        IF (.NOT. STATUS) THEN
                                WRITE (6,*) 'Invalid date format'
                        ELSE
                                valid_date = .TRUE.

                        END IF
                END DO
```

15.2.2 Using Literals

Use literal values to replace variables in the same way you would in any FORTRAN program. Literal values can be either numeric or character strings. String literals must be quoted in single quotation marks (' ') or double quotation marks (" ") in FORTRAN (although they are always printed as single quotation marks). You may use a literal in any Rdb/VMS data manipulation statement that accepts a host language variable.

```
&RDB&  FOR D IN DEPARTMENTS WITH
&RDB&    D.DEPARTMENT_CODE = 'ADMN'
&RDB&    GET
&RDB&      DEP_NAME = D.DEPARTMENT_NAME
&RDB&    END_GET
&RDB&  END_FOR
```

15.2.3 Forming Record Streams

In FORTRAN, and any language that you use to access an Rdb/VMS database, you select the records you are interested in manipulating by gathering these records into a stream. You create this stream using the Rdb/VMS data manipulation statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

15.2.4 Retrieving Records

Rdb/VMS provides you with three statements to retrieve records:

- FOR
- Two START_STREAM statements:
  - Declared START_STREAM
  - Undeclared START_STREAM

The following sections provide FORTRAN examples of how to form record streams and retrieve records using the FOR and START_STREAM statements.

15.2.4.1 Using the FOR Statement to Retrieve Records The FOR statement forms a record stream and provides automatic iteration for any Rdb/VMS and FORTRAN statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

Example 15–5 shows a FOR statement from the F_DISPLAY_CAND.RFO subroutine. It uses the flag "success" to determine if the RSE has been satisfied. If a candidate record is found with field values that match the values in the host language variables, the success flag is set to true. If no record matches the values in the host language variables, then the success flag remains set to false.

Example 15–5 Using the FOR Statement in RDBPRE FORTRAN

```
                success = .FALSE.
&RDB&           START_TRANSACTION READ_ONLY
&RDB&               FOR C IN CANDIDATES WITH
&RDB&                   C.FIRST_NAME = candidate_name AND
&RDB&                   C.MIDDLE_INITIAL = candidate_mi AND
&RDB&                   C.LAST_NAME = candidate_lname

&RDB&                   GET candidate_status = C.CANDIDATE_STATUS
&RDB&                   END_GET
                        success = .TRUE.
                        TYPE 4000,candidate_name,
        1                       candidate_mi,candidate_lname
4000                    FORMAT('/',A,' ',A,' ',A,' has the
        1following status: ')
                        TYPE *, candidate_status
&RDB&               END_FOR
```

You can include host language statements within the FOR . . . END_FOR block to process the records within the stream. However, there is an important exception to the type of statement you can include. Do not transfer control out of the FOR . . . END_FOR block unless you do not want to return. It is impossible to enter the loop again while it is executing.

You may call a module from within a FOR loop, because these subroutines execute within the FOR loop context. However, you cannot use a context variable defined in the FOR block in any subroutine that is preprocessed outside the FOR block.

15.2.4.2 Using Declared Streams to Retrieve Records Rdb/VMS supports two forms of the START_STREAM statement. The *declared* START_STREAM statement and the *undeclared* START_STREAM statement. Declared streams provide all the features of the undeclared streams and more. Most importantly, undeclared streams require that the statements you use to manipulate the stream be enclosed by the START_STREAM and END_STREAM statements in your source program. Declared streams do not impose this restriction. The statements you use to manipulate the stream may appear in any order within your program as long as the DECLARE_STREAM statement appears first and the statements execute in a logical order (START_STREAM, FETCH, GET, END_STREAM).

Digital recommends that all new applications use the declared START_ STREAM statement. For this reason, only the declared START_STREAM statement is discussed in this section. Complete details on the differences between declared and undeclared START_STREAM statements are provided in Chapter 9.

Note *If you use the AT END clause with a FETCH clause, you must use the END_ FETCH clause to terminate the FETCH statement.*

Example 15–6, from the F_PAIR.RFO subroutine, shows the use of the declared START_STREAM and FETCH statements. The example pairs a CANDIDATES record with an EMPLOYEES record at random. This could not be achieved with a FOR statement. You could not conditionally end a FOR loop when all the CANDIDATES records have been paired with EMPLOYEES records. A START_STREAM statement lets you do this.

Example 15–6 Using the Declared START_STREAM and FETCH Statements in RDBPRE FORTRAN

```
        SUBROUTINE pair
C---------------------------------------------------------
C This subroutine demonstrates the use of the declared
C START_STREAM statement.  The output of this
C subroutine is merely a random matching of each CANDIDATES
C record with an EMPLOYEES record.
C----------------------------------------------------------

        IMPLICIT NONE
        LOGICAL emps_end,cands_end
        CHARACTER employee_id*5,last_name*14,first_name*10,confirm
        CHARACTER cand_last_name*14,cand_first_name*10,cand_status*257

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

        WRITE (6,90)
90      FORMAT ('1',T25,'**** EMPLOYEES/CANDIDATES ****'///)

C--------------------------------------------------------
C Declare the streams that will be used to process the
C EMPLOYEES and CANDIDATES records.
C--------------------------------------------------------

&RDB&   DECLARE_STREAM cands USING ca IN candidates
&RDB&           SORTED BY ca.last_name
&RDB&   DECLARE_STREAM emps USING em IN employees
&RDB&           SORTED BY em.first_name

&RDB&   START_TRANSACTION READ_ONLY
C-------------------------------------------------------
C Open both streams and set a flag for the
C end-of-stream condition to false.
C-------------------------------------------------------

&RDB&       START_STREAM cands
&RDB&       START_STREAM emps

                emps_end = .FALSE.
                cands_end = .FALSE.

C--------------------------------------------------------
C Fetch a record from the CANDIDATES relation.  If there
C are no records to retrieve, set the end-of-stream flag
C to true.  Otherwise, retrieve the record.
C--------------------------------------------------------
```

(continued on next page)

Example 15–6 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE FORTRAN

```
&RDB&           FETCH cands
&RDB&                   AT END
                                cands_end = .TRUE.
&RDB&           END_FETCH
                IF (.NOT.(cands_end)) THEN
&RDB&                   GET
&RDB&                           cand_last_name = ca.last_name;
&RDB&                           cand_first_name = ca.first_name;
&RDB&                           cand_status = ca.candidate_status;
&RDB&                   END_GET
                END IF

C--------------------------------------------------------
C Fetch a record from the EMPLOYEES relation.  If there
C are no records to retrieve, set the end-of-stream flag
C to true.  Otherwise, retrieve the record.
C--------------------------------------------------------

&RDB&           FETCH emps
&RDB&                   AT END
                                emps_end = .TRUE.
&RDB&           END_FETCH
                IF (.NOT.(emps_end)) THEN
&RDB&                   GET
&RDB&                           last_name = EM.LAST_NAME;
&RDB&                           first_name = EM.FIRST_NAME;
&RDB&                           employee_id = EM.EMPLOYEE_ID;
&RDB&                   END_GET
                END IF
C-------------------------------------------------------
C If a record exists in both relations then print a
C report.
C-------------------------------------------------------
```

(continued on next page)

Example 15–6 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDBPRE FORTRAN

```
                PRINT *,' Employees:                 Candidates:'
                PRINT *,'-------------------------  -------------------------'
                DO WHILE (.NOT.(cands_end))
                        TYPE 100,last_name,first_name,
        1                        cand_last_name,cand_first_name
100                     FORMAT('/',' ',A,' ',A,T31,A,' ',A)
&RDB&                   FETCH cands
&RDB&                           AT END
                                        cands_end = .TRUE.
&RDB&                   END_FETCH
                        IF (.NOT.(cands_end)) THEN
&RDB&                           GET
&RDB&                                   cand_last_name = CA.LAST_NAME;
&RDB&                                   cand_first_name = CA.FIRST_NAME;
&RDB&                                   cand_status = CA.CANDIDATE_STATUS;
&RDB&                           END_GET
                        END IF
                        IF (.NOT.(emps_end)) THEN
&RDB&                           FETCH emps
&RDB&                                   AT END
                                                emps_end = .TRUE.
&RDB&                           END_FETCH
                                IF (.NOT.(emps_end)) THEN
&RDB&                               GET
&RDB&                                   last_name = EM.LAST_NAME;
&RDB&                                   first_name = EM.FIRST_NAME;
&RDB&                                   employee_id = EM.EMPLOYEE_ID;
&RDB&                               END_GET
                                END IF
                        END IF
                END DO
C-----------------------------------------------
C Close both streams.
C-----------------------------------------------

&RDB&           END_STREAM emps
&RDB&           END_STREAM cands

&RDB&   COMMIT

        PRINT *, '  '
        TYPE 300
300     FORMAT ('$',' Press RETURN to continue')
        ACCEPT 400, confirm
400     FORMAT (A)

        RETURN
        END
```

15.2.5 Retrieving Segmented Strings

Retrieving segmented strings is a two-step process. First, you must retrieve the record that contains the segmented string field; then, you must retrieve the individual segments that comprise the segmented string field.

You may find it easier to picture a segmented string by referring to Figure 8–1 in Chapter 8.

Rdb/VMS provides you with two statements to retrieve segmented string fields:

- FOR
- START_SEGMENTED_STRING

15.2.5.1 Using the FOR Statement to Retrieve Segmented Strings You must use two streams when processing segmented string streams. Use the first FOR or START_STREAM statement to form an outer stream of records, and then use the second FOR statement to form an inner stream of segments. The inner stream formed by the second RSE identifies the segments contained in the field specified by the outer stream formed by the first RSE. Use different context variables in the inner and outer streams.

Remember that to retrieve the segmented string, you must begin at the first segment and retrieve segments in the order that they are stored, that is, sequentially.

Example 15–7 from the F_DISPLAY_RESUME.RFO subroutine:

- Uses a FOR statement to search the database for a record with a value for the EMPLOYEE_ID field that matches the host language variable, employee_id
- Uses a second FOR statement to loop through the segments of the segmented string field for the EMPLOYEES record
- Uses the GET statement to retrieve the individual segments that comprise a segmented string
- Displays these values on the terminal

Example 15–7 Using the FOR Statement with Segmented Strings in RDBPRE FORTRAN

```
        SUBROUTINE display_resume

C-----------------------------------------------------
C This subroutine demonstrates how to retrieve a
C field of data type SEGMENTED STRING.
C-----------------------------------------------------

        IMPLICIT NONE
        LOGICAL employee_found
        INTEGER*4 segment_length
        CHARACTER employee_id*5,resume_segment*80

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

        WRITE (6,90)
90      FORMAT ('1',T25,'**** DISPLAY RESUME ****'///)

C-------------------------------------------------
C Prompt user to enter the ID of the employee
C resume that he or she wants to view.  If user
C enters 'exit' then exit subroutine.
C-------------------------------------------------
        TYPE 110
110     FORMAT ('$',' Please enter the ID of the
        1 Employee or type exit: ')
        ACCEPT 120, employee_id
120     FORMAT (A)

        DO WHILE ((employee_id.NE.'EXIT ').AND.(employee_id.NE.'exit '))

                employee_found = .FALSE.
&RDB&           START_TRANSACTION READ_ONLY

C----------------------------------------------------
C Start an outer FOR loop to retrieve the employee
C record(s) with the specified ID.
C-----------------------------------------------------

&RDB&                   FOR R IN RESUMES WITH
&RDB&                           R.EMPLOYEE_ID = employee_id
                                employee_found = .TRUE.

C------------------------------------------------------
C Start an inner FOR loop to retrieve the segments
C of the segmented string that comprise the employee's
C resume.
C------------------------------------------------------

&RDB&                           FOR RR IN R.RESUME
&RDB&                           GET
&RDB&                               resume_segment = RR.RDB$VALUE;
&RDB&                               segment_length = RR.RDB$LENGTH;
&RDB&                           END_GET
```

(continued on next page)

Example 15–7 (Cont.) Using the FOR Statement with Segmented Strings in RDBPRE FORTRAN

```
C--------------------------------------------------
C Display each segment as it is retrieved from
C the database.
C--------------------------------------------------

                                TYPE 1000, resume_segment
1000                            FORMAT (' ',A)
&RDB&                           END_FOR
&RDB&                   END_FOR
&RDB&           COMMIT

C--------------------------------------------------
C If a record with the specified ID was not found
C then inform the user.
C--------------------------------------------------

                IF (.NOT.(employee_found)) THEN
                        TYPE 2000, employee_id
2000                    FORMAT('  Employee ',A,' has no resume on file')
                END IF

                PRINT *, '  '
                TYPE 110
                ACCEPT 120, employee_id

        END DO

        RETURN
        END
```

The GET statement fetches only as much of the stored segment as the host language variable that receives the segment can hold. The next GET statement fetches the next piece of the segment. Suppose the segmented string segment size in the previous example was declared as 80 characters and the actual length of the stored segment was 100 characters. The first GET statement would fetch 80 characters of the first segment and the next GET statement would fetch the remaining 20 characters. The third GET statement would fetch 80 characters of the second segment, the next GET statement would fetch the remaining 20, and so on.

15.2.5.2 Using the START_SEGMENTED_STRING Statement to Retrieve Segmented Strings When you want to maintain program control of loop iteration through the segments that form a segmented string, use the START_SEGMENTED_STRING statement with a record stream formed by a FOR or START_STREAM statement. You must start two streams when processing segmented string streams with the START_SEGMENTED_STRING statement.

Form an outer stream of records with a FOR or START_STREAM statement, then use the START_SEGMENTED_STRING statement to form an inner stream of segments. This inner stream identifies the segment stream that is contained in the field specified by the FOR or START_STREAM statement. When you name the segment stream, use a different name from the outer stream name. Use different context variables for the outer stream and the inner segmented string stream.

The program shown in Example 15–8:

- Uses an undeclared START_STREAM statement to find all the records in the RESUMES relation with an employee ID of 12345.
- Uses a START_SEGMENTED_STRING statement to retrieve the resume of each EMPLOYEES record found by the first stream.
- Uses the GET statement to retrieve the segments that comprise the segmented string.
- Checks the return status value of the GET statement after each segment is retrieved to make sure the end-of-segmented-string condition has not been met. If this condition has not been met, the value of the current segment is printed.
- Stops processing the segmented string field when the preceding condition is met.
- Fetches the next employee record with an employee ID of 12345, if one exists.
- Closes both streams when both the START_STREAM and START_SEGMENTED_STRING end conditions have been met.
- Commits the transaction.

Example 15–8 Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE FORTRAN

```
        PROGRAM show_resume

        IMPLICIT NONE
        LOGICAL end_of_stream
        INTEGER*4 segment_length,status
        CHARACTER resume_segment*80
        INTEGER*4 RDB$_SEGSTR_EOF
        EXTERNAL  RDB$_SEGSTR_EOF
        EXTERNAL RDB$SIGNAL

&RDB&   DATABASE pers=FILENAME 'MF_PERSONNEL'
        end_of_stream = .FALSE.
&RDB&           START_TRANSACTION READ_ONLY
```

(continued on next page)

Example 15–8 (Cont.) Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE FORTRAN

```
C----------------------------------------------
C Find all the records in the RESUMES relation
C with an employee ID of 12345.
C----------------------------------------------

&RDB&               START_STREAM RESSTR USING
&RDB&                 R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
&RDB&                     FETCH RESSTR

&RDB&                     END_FETCH
C-----------------------------------------------
C Retrieve the resume of each employee found
C with the START_STREAM statement.
C-----------------------------------------------

&RDB&               START_SEGMENTED_STRING RINFO USING STRN IN R.RESUME
                         DO WHILE (.NOT. (end_of_stream))

C--------------------------------------------------
C Retrieve the segments that comprise the segmented
C string field.
C--------------------------------------------------

&RDB&                       GET
&RDB&                         ON ERROR
                                CALL RDB$SIGNAL( )
&RDB&                         END_ERROR
&RDB&                           resume_segment = STRN.RDB$VALUE;
&RDB&                           segment_length = STRN.RDB$LENGTH;
&RDB&                       END_GET

C----------------------------------------------------
C Check the return status value of the GET statement
C after each segment is retrieved to make sure that
C the end-of-segmented-string condition has not
C been met.  If this condition has not been met,
C print the value of the current segment.  Otherwise,
C stop processing the stream of segments.
C----------------------------------------------------

                              status = RDB$MESSAGE_VECTOR(2)

                       IF (status .NE. (%LOC(RDB$_SEGSTR_EOF))) THEN
                                   TYPE 2000, resume_segment
2000                               FORMAT (' ',A)
                              ELSE
                                end_of_stream = .TRUE.
                              END IF

                         END DO
```

(continued on next page)

Example 15–8 (Cont.) Using the START_STREAM and START_SEGMENTED_STRING Statements in RDBPRE FORTRAN

```
C--------------------------------------------------
C Close both streams.
C--------------------------------------------------

&RDB& END_SEGMENTED_STRING RINFO
&RDB& END_STREAM RESSTR
&RDB&          COMMIT
        STOP
        END
```

15.2.6 Retrieving Field Values

Use the GET statement to retrieve one, several, or all the field values from a database record. You can also use the GET statement to retrieve statistical values from the database.

Do not use the RDBPRE concatenation operator (|) in a GET statement. Doing so causes a preprocessing error. To concatenate fields in preprocessed programs, first use the GET statement to retrieve the individual fields and store them in separate FORTRAN variables. Then concatenate the FORTRAN variables in a FORTRAN statement using the FORTRAN concatenation operator, double slashes (//).

Section 15.2.6.1 and Section 15.2.6.2 provide examples of retrieving field and record values. Section 15.2.6.3 provides an example of retrieving statistical values.

15.2.6.1 Using the GET Statement to Retrieve Field Values When you form a record stream using the FOR statement, you include the GET statement within the FOR . . . END_FOR block to retrieve field values from the record stream. When you form a record stream using the undeclared START_STREAM statement, you include the GET statement between the START_STREAM and END_STREAM statements. When you use the declared form of the START_STREAM statement, the GET statement must execute within the START_STREAM . . . END_STREAM block; however, it does not have to appear within this block in your program.

Example 15–9, from the F_LIST_RECORD.RFO subroutine, shows the use of the FOR and GET statements in RDBPRE FORTRAN.

Example 15–9 Using the FOR and GET Statements in RDBPRE FORTRAN

```
C-------------------------------------------------------
C For each EMPLOYEES record that has a corresponding
C record in DEGREES, print the DEGREES record.
C-------------------------------------------------------

&RDB&           FOR E IN EMPLOYEES SORTED BY E.LAST_name
&RDB&               FOR D IN DEGREES WITH
&RDB&                   D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                   GET
&RDB&                           last_name = E.LAST_NAME;
&RDB&                           first_name = E.FIRST_NAME;
&RDB&                           degree = D.DEGREE;
&RDB&                           degree_field = D.DEGREE_FIELD;
&RDB&                   END_GET
                    TYPE 100,first_name,last_name,degree,degree_field
100                 FORMAT('/',' Name is: ',A,' ',A,/,'   Degree
        1is: ',A,'   Degree field is: ',A/)
&RDB&               END_FOR
                     .
                     .
                     .
&RDB&           END_FOR
```

See an earlier example, Example 15–6, for a demonstration of how to use the START_STREAM, FETCH, and GET statements.

15.2.6.2 Using the GET * Statement to Retrieve Field Values A special form of the GET statement is the GET * statement, which lets you retrieve database values at the record level rather than the field level. You can retrieve all the fields in a record with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the records of a relation, with record field names that match the database field names. You can use the FORTRAN DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) The GET * statement in the following example retrieves all the fields from the records of the JOB_HISTORY relation and places their values in the job_history host language record structure:

```
&RDB& FOR FIRST 1 J IN JOB_HISTORY WITH
&RDB&   J.JOB_CODE = JOB_CODE IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   GET
&RDB&     job_history = J.*
&RDB&   END_GET
&RDB& END_FOR
```

15.2.6.3 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly, without processing each record in the record stream. RDBPRE may assign a data type to the result that is different from the data type of the field referred to in the expression. See Chapter 8 for information on the data type conversions performed by statistical expressions.

Example 15–10, from the F_STATS.RFO subroutine, uses the statistical function COUNT to find the total number of records in the EMPLOYEES relation.

Example 15–10 Using the GET Statement to Retrieve Statistical Values in RDBPRE FORTRAN

```
        SUBROUTINE stats
C----------------------------------------------
C This subroutine displays the total
C number of records stored in the EMPLOYEES
C relation.
C----------------------------------------------

        IMPLICIT NONE
        INTEGER number_employees
        CHARACTER confirm

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

        WRITE (6,90)
90      FORMAT ('1',T25,'**** STATISTICS ****'///)

C-----------------------------------------------------
C Use the GET statement with a statistical function
C to calculate the total number of records in the
C EMPLOYEES relation.
C-----------------------------------------------------

&RDB&   START_TRANSACTION READ_ONLY
&RDB&   GET number_employees = COUNT OF e IN employees END_GET
C-----------------------------------------
C Display the value.
C-----------------------------------------

        TYPE 100, number_employees
100     FORMAT ('/',' Number of employees in the
        1Corporation are: ',I5,/)
&RDB&   COMMIT
```

(continued on next page)

Example 15–10 (Cont.) Using the GET Statement to Retrieve Statistical Values in RDBPRE FORTRAN

```
        PRINT *, '  '
        TYPE 300
300     FORMAT ('$',' Press RETURN to continue')
        ACCEPT 400, confirm
400     FORMAT (A)

        RETURN
        END
```

15.2.7 Updating Records Using the STORE, MODIFY, and ERASE Statements

The Rdb/VMS update statements can only be used in a read/write transaction. (You may, of course, include any valid Rdb/VMS statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

If you update a record and triggered actions have been defined for the relation containing the record, the update operation (STORE, MODIFY, or ERASE) will have the specified effect on all the relations in the database that have a foreign key relationship with the record you want to update.

If a relation-specific constraint has been defined, your ability to perform update operations may depend on the presence of matching field values in other relations. For more information on relation-specific constraints, see Section 6.6.

Include the GET statement in a read/write transaction if you intend to update any of the fields returned by the GET statement.

Note *You may not use a view to update records if that view refers to more than one relation.*

15.2.7.1 Storing Records You can insert values into one or more fields in one relation using a single STORE statement. To store more than one record in a relation, include the STORE statement within a program loop.

Example 15–11, from the F_ADD_EMPLOYEES.RFO subroutine, stores an employee record in the EMPLOYEES relation.

Example 15–11 Storing Records in RDBPRE FORTRAN

```
                 .
                 .
                 .
                success = .FALSE.
                retry_count = 0

            DO WHILE ((retry_count .LT. 5) .AND. (.NOT. (success)))
                success = .TRUE.
&RDB&           START_TRANSACTION READ_WRITE NOWAIT RESERVING
&RDB&                   EMPLOYEES FOR SHARED WRITE
&RDB&           ON ERROR
                        success = .FALSE.
                        retry_count = retry_count + 1
                        CALL error_handler(RDB$STATUS,success)
                        IF (success) THEN
                                retry_count = 5
                        END IF
&RDB&           END_ERROR
            END DO

                success = .FALSE.
                retry_count = 0

C------------------------------------------------------------
C The following loop will execute at least once, because
C 'success' has just been set to false, and 'retry_count' to
C zero.  If an error occurs during the STORE operation, the
C program will retry STORE operation up to 5 times.
C-------------------------------------------------------------

            DO WHILE ((retry_count .LT. 5) .AND. (.NOT. (success)))
                success = .TRUE.
&RDB&           STORE E IN EMPLOYEES USING
&RDB&           ON ERROR
                        success = .FALSE.
                        retry_count = retry_count + 1
                        CALL error_handler(RDB$STATUS,success)
                        IF (success) THEN
                                retry_count = 5
                        END IF
&RDB&           END_ERROR
```

(continued on next page)

Example 15–11 (Cont.) Storing Records in RDBPRE FORTRAN

```
C--------------------------------------------------------
C Store the values that the user entered in an
C EMPLOYEES record.
C--------------------------------------------------------
&RDB&                  E.EMPLOYEE_ID = employee_id;
&RDB&                  E.LAST_NAME = last_name;
&RDB&                  E.FIRST_NAME = first_name;
&RDB&                  E.MIDDLE_INITIAL = middle_initial;
&RDB&                  E.ADDRESS_DATA_1 = address_data_1;
&RDB&                  E.ADDRESS_DATA_2 = address_data_2;
&RDB&                  E.CITY = city;
&RDB&                  E.STATE = state;
&RDB&                  E.POSTAL_CODE = postal_code;
&RDB&                  E.BIRTHDAY = birthday;
                          .
                          .
                          .
&RDB&          END_STORE
            END DO
```

15.2.7.1.1 Using the STORE * Statement to Store Records A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in a record with the STORE * statement. To use the STORE * statement, you must first declare a record structure that contains all the fields in the relation, with record field names that match the database field names. You can use the FORTRAN DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to store in the record fields and store the entire record using the STORE * statement. Example 15–12 shows the use of the STORE * statement to store job_history, a host language record structure, in the JOB_HISTORY relation.

Example 15–12 Using the STORE * Statement in RDBPRE FORTRAN

```
&RDB& STORE J IN PERS.JOB_HISTORY USING
&RDB&   J.* = job_history;
&RDB& END_STORE
```

15.2.7.1.2 Using the CREATE_SEGMENTED_STRING Statement to Store Segmented Strings

Use the CREATE_SEGMENTED_STRING statement and the STORE statement to store segmented strings in a relation. You must use two operations to store segmented strings.

Note *See Section 9.2.6.1.2 for information about defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name with an appropriate value for storing your segmented strings.*

Note *Segmented strings cannot be updated (ERASE, MODIFY, or STORE) as part of a triggered action. For more information, see the DEFINE TRIGGER statement in the* VAX Rdb/VMS RDO and RMU Reference Manual.

Example 15–13, from the F_MOD_RESUME.RFO subroutine, demonstrates how to read and store a resume into a segmented string from a sequential file; then it shows how to use the segmented string handle to modify an existing database record.

Example 15–13 Using the CREATE_SEGMENTED_STRING Statement in RDBPRE FORTRAN

```
       SUBROUTINE store_res
C--------------------------------------------------------------
C This subroutine demonstrates how to store a record with a
C field of data type SEGMENTED STRING.
C--------------------------------------------------------------

       IMPLICIT NONE
       LOGICAL end_of_file
       CHARACTER employee_id*5,resume_line*80,resume_file*30

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

       WRITE (6,90)
90     FORMAT ('1',T25,'**** MODIFY RESUME ****'///)
```

(continued on next page)

Example 15–13 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in RDBPRE FORTRAN

```
C-------------------------------------------------------
C Prompt user for the employee ID of the employee
C resume that he or she wants to store.
C-------------------------------------------------------

100     TYPE 110
110     FORMAT ('$',' Please enter the ID of the
        1Employee or type exit: ')
        ACCEPT 120, employee_id
120     FORMAT (A)

        DO WHILE ((employee_id.NE.'EXIT ').AND.(employee_id.NE.'exit '))

                PRINT *, '  '
C----------------------------------------------------------
C Prompt user for the file name of the resume to be stored.
C----------------------------------------------------------

                TYPE 1006
1006            FORMAT ('$','   Please enter filename of new resume: ')
                ACCEPT 1010, resume_file
1010            FORMAT (A)

&RDB&           START_TRANSACTION READ_WRITE
&RDB&                   RESERVING RESUMES FOR SHARED WRITE

C---------------------------------------------------------
C Create a segmented string to hold the values from the
C specified file.
C---------------------------------------------------------

&RDB&                   CREATE_SEGMENTED_STRING resume_handle

                        end_of_file = .FALSE.
                        OPEN (UNIT=1, FILE=resume_file, STATUS='old')

                    DO WHILE (.NOT.(end_of_file))
                        READ (1, 2000, END=3000) resume_line
2000                    FORMAT (A80)
&RDB&                   STORE R IN resume_handle USING
&RDB&                           R.RDB$VALUE = resume_line
&RDB&                   END_STORE
                    END DO

3000                    end_of_file = .TRUE.
                        CLOSE (UNIT=1)

&RDB&                   END_SEGMENTED_STRING resume_handle
```

(continued on next page)

Example 15–13 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in RDBPRE FORTRAN

```
C-----------------------------------------------------------------
C Store the new record by supplying the segmented string handle from
C the CREATE_SEGMENTED_STRING statement as the object of the segmented
C string assignment statement.
C-----------------------------------------------------------------

&RDB&                   STORE R IN RESUMES USING
&RDB&                           R.EMPLOYEE_ID = employee_id;
&RDB&                           R.RESUME = resume_handle;
&RDB&                   END_STORE
&RDB&           COMMIT

                PRINT *, '  '
                TYPE 110
                ACCEPT 120, employee_id

        END DO

        RETURN
        END
```

15.2.7.2 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of a record in a relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Example 15–14, a FORTRAN program segment from the F_MODIFY_ ADDRESS.RFO subroutine, modifies a record in the EMPLOYEES relation.

Example 15–14 Modifying Records in RDBPRE FORTRAN

```
                     .
                     .
                     .
&RDB&                   START_TRANSACTION READ_WRITE RESERVING
&RDB&                           EMPLOYEES FOR SHARED WRITE

C-----------------------------------------------
C Start a record stream containing records
C with an employee identification number equal
C to the host language variable 'employee_id'.
C-----------------------------------------------

&RDB&                   FOR E IN EMPLOYEES WITH
&RDB&                           E.EMPLOYEE_ID = employee_id

C-----------------------------------------------------
C Modify the records in the record stream.  If an
C error occurs during the MODIFY operation, call an
C error handler.
C-----------------------------------------------------

&RDB&                       MODIFY E USING
&RDB&                       ON ERROR
                                success = .FALSE.
                                CALL error_handler(RDB$STATUS,success)
                                IF (success) THEN
                                        retry_count = 5
                                END IF
&RDB&                       END_ERROR

C----------------------------------------------------------------
C If no error has occurred, change the value of the following
C fields to the values in the host language variables.
C Host language variables are in lowercase.
C----------------------------------------------------------------

&RDB&                           E.ADDRESS_DATA_1 = address_data_1;
&RDB&                           E.ADDRESS_DATA_2 = address_data_2;
&RDB&                           E.CITY = city;
&RDB&                           E.STATE = state;
&RDB&                           E.POSTAL_CODE = postal_code;
&RDB&                       END_MODIFY
&RDB&                   END_FOR
```

(continued on next page)

Example 15–14 (Cont.) Modifying Records in RDBPRE FORTRAN

```
C-------------------------------------------------
C If the value of the success flag is TRUE, the
C modify operation was successful, so commit the
C transaction.  Otherwise the modify operation was
C not successful, so roll back the active
C transaction.
C--------------------------------------------------

                        IF (success) THEN
                                PRINT *, ' Update operation successful'
&RDB&                           COMMIT
                        ELSE
                                PRINT *, ' Update operation failed'
&RDB&                           ROLLBACK
                        END IF
```

15.2.7.2.1 Using the MODIFY * Statement to Modify Records A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. You can use the FORTRAN DICTIONARY statement to create such a record structure. (See Chapter 12 for more information on copying record and field definitions from the data dictionary.) Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement.

Only use the MODIFY * statement if you need to modify every field value in a record. Modifying a field by replacing one value with an identical value needlessly adds overhead to your program. For example, your program may check constraints on a field value that *you know* is valid because it is the same value that the field presently holds.

Example 15–15 replaces the field values of an employee record in the JOB_HISTORY relation with the values in the job_history host language record structure.

Example 15–15 Using the MODIFY * Statement in RDBPRE FORTRAN

```
&RDB& FOR J IN JOB_HISTORY WITH
&RDB&   J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
&RDB&     AND J.JOB_END MISSING
&RDB&   MODIFY J USING
&RDB&     J.* = job_history;
&RDB&   END_MODIFY
&RDB& END_FOR
```

15.2.7.2.2 Modifying Segmented Strings To modify a segmented string, you must first create a new segmented string with the CREATE_SEGMENTED_STRING statement and then modify the existing record by replacing the logical pointer to the old segmented string with the logical pointer to the new segmented string. You accomplish this by using the segmented string handle in an assignment statement. As Chapter 8 explains in more detail, when you store a segmented string field, you do not actually store segments into a record; rather, you store a logical pointer to the first segment in the segmented string. Thus, by creating a new segmented string and a new segmented string id associated with it, you can modify the field in a database record that "contains" a segmented string merely by replacing the old segmented string id with a new segmented string id. When you use the segmented string handle in an assignment statement, RDBPRE understands that it is the segmented string id that is to be assigned to the record.

Note *Although you use a MODIFY statement to modify segmented strings, you are not actually modifying the individual segments that comprise the segmented string field. You are actually replacing the entire segmented string with a new segmented string.*

See an earlier example, Example 15–13, for an illustration of how this is done in FORTRAN.

15.2.7.3 Erasing Records You can delete one, many, or all the records from a relation using a single ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase.

Example 15–16, from the F_DELETE_RECORD.RFO subroutine, demonstrates how to ERASE records in FORTRAN programs.

Note *The definition of the sample personnel database includes the trigger EMPLOYEE_ID_CASCADE_DELETE, which performs an automatic deletion of records in the relations named in ERASE statements in Example 15–16 (except for RESUMES) when the record with the matching employee ID is deleted from the EMPLOYEES relation. Thus, you would not need to include "cascading deletion" logic in your programs if it were already included in a trigger definition.*

Example 15–16 Erasing Records in RDBPRE FORTRAN

```
&RDB&           START_TRANSACTION READ_WRITE RESERVING
&RDB&               EMPLOYEES, SALARY_HISTORY, JOB_HISTORY,
&RDB&               DEPARTMENTS, DEGREES, WORK_STATUS,
&RDB&               RESUMES FOR SHARED WRITE
&RDB&               FOR E IN EMPLOYEES WITH
&RDB&                   E.RDB$DB_KEY = data_base_key
&RDB&                   FOR JH IN JOB_HISTORY WITH
&RDB&                       JH.EMPLOYEE_ID = e.employee_id
&RDB&                       ERASE JH
&RDB&                   END_FOR
&RDB&                   FOR SH IN SALARY_HISTORY WITH
&RDB&                       SH.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                       ERASE SH
&RDB&                   END_FOR
&RDB&                   FOR D IN DEGREES WITH
&RDB&                       D.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                       ERASE D
&RDB&                   END_FOR
&RDB&                   FOR R IN RESUMES WITH
&RDB&                       R.EMPLOYEE_ID = E.EMPLOYEE_ID
&RDB&                       ERASE R
&RDB&                   END_FOR
&RDB&                   ERASE E
&RDB&               END_FOR
&RDB&           COMMIT
```

15.3 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer or address that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this key as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, the scope of a dbkey ends with a COMMIT statement. That is, a dbkey is guaranteed to point to the same record for the life of the transaction in which it is retrieved.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

The following example demonstrates how to specify the scope of the dbkey in an RDBPRE FORTRAN program:

```
&RDB&   DATABASE GLOBAL pers = FILENAME 'MF_PERSONNEL'

C--------------------------------------------------
C Extend the scope of the database key so that
C it will be valid across transactions.
C--------------------------------------------------

&RDB&   DBKEY SCOPE IS FINISH
```

Suggestions on how you can take advantage of the dbkey scope are contained in Section 9.2.7.

15.4 Using Structured Programming

Programs and modules that pass through the RDBPRE preprocessor do not have unlimited freedom in structure.

Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- DECLARE_STREAM
- FOR
- GET
- START_STREAM
- END_STREAM
- FETCH
- STORE
- MODIFY
- ERASE
- CREATE_SEGMENTED_STRING
- START_SEGMENTED_STRING
- END_SEGMENTED_STRING

Each of these individual data manipulation statements forms only part of a complex call to the database. The preprocessor generates one call to the database, using more than one data manipulation statement. For example, a MODIFY statement executes within the context of a FOR or START_STREAM statement. The call to the database can only be made using both the FOR and MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. However, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

Also keep in mind that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement, and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which it is issued.

A stream declared with the DECLARE_STREAM statement lets you place the stream of manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

For more information on the declared and undeclared START_STREAM statement, see Section 9.2.3.2. Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- DATABASE
- READY
- START_TRANSACTION
- GET
- COMMIT
- ROLLBACK
- FINISH
- DECLARE_STREAM

Remember that you must issue the DECLARE_STREAM statement before you can issue a declared START_STREAM statement, and the DATABASE statement must appear in the data declaration section of your program.

Note *You must preprocess FORTRAN functions, subroutines, or submodules in separate files. Once the individual files have been successfully preprocessed, you can easily obtain a single executable image. Link the submodule object (OBJ)* *file or files with the main object module to create one executable (EXE) program image.*

Example 15–17, from the F_DELETE_RECORD.RFO and the F_CALL_OTHER.RFO subroutines, demonstrates structured programming in a preprocessed FORTRAN program. The F_DELETE_RECORD.RFO module and the F_CALL_OTHER subroutine are separately preprocessed and compiled. They are linked with the LINK command. The F_DELETE_RECORD module passes the value of the dbkey to the F_CALL_OTHER subroutine. This subroutine finds the record associated with the dbkey and displays this record on the terminal. Although it is not necessary to program this query in two modules, it is done here to demonstrate how to pass variables between separately compiled modules.

Example 15–17 Using Data Manipulation Statements in Structured Programming in RDBPRE FORTRAN

```
MODULE: F_DELETE_RECORD:

&RDB&           START_TRANSACTION (TRANSACTION_HANDLE trans1)
&RDB&               READ_WRITE RESERVING EMPLOYEES FOR SHARED READ

                        found_employee = .FALSE.
                        success = .TRUE.

C-------------------------------------------------------
C Find the employee record that the user wants to
C delete.  If an error occurs during the FOR operation,
C call an error handler.
C---------------------------------------------------------

&RDB&                   FOR (TRANSACTION_HANDLE trans1)
&RDB&                       E IN EMPLOYEES WITH
&RDB&                           E.EMPLOYEE_ID = employee_id
&RDB&                       ON ERROR
                                success = .FALSE.
                                CALL error_handler(RDB$STATUS,success)
                                IF (success) THEN
                                        retry_count = 5
                                END IF
&RDB&                           END_ERROR
```

(continued on next page)

Example 15–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE FORTRAN

```
C-------------------------------------------------------
C Get the database key of the EMPLOYEES record that the
C user wants to delete.
C-------------------------------------------------------

&RDB&                   GET
&RDB&                       ON ERROR
                                success = .FALSE.
&RDB&                       END_ERROR
&RDB&                       data_base_key = E.RDB$DB_KEY
&RDB&                   END_GET
                        found_employee = .TRUE.
&RDB&                   END_FOR

                        IF (.NOT.(found_employee)) THEN
                                TYPE 2020, employee_id
2020                            FORMAT (' Employee id: ',A,' is
        1 not on file')

C-------------------------------------------------------------
C Pass the dbkey to an external routine CALL_OTHER
C to print out the record to which the dbkey points.
C Note that using an external routine is neither necessary
C nor recommended for performing this task.  It is done in this
C example only to show how values are passed between routines
C in an RDBPRE FORTRAN program.
C-------------------------------------------------------------

                        ELSE IF (success) THEN
                                CALL call_other(data_base_key,trans1)
                        END IF
&RDB&           COMMIT (TRANSACTION_HANDLE trans1)
```

Subroutine CALL_OTHER:

```
        SUBROUTINE call_other(data_base_key,trans1)

C-----------------------------------------------------
C This subroutine is passed the dbkey and transaction
C handle from the DELETE_RECORD subroutine.
C With this information, it can find and display the
C employee record associated with an employee_id specified
C in DELETE_RECORD and then return program control to the
C DELETE_RECORD subroutine.
C-------------------------------------------------------

        IMPLICIT NONE
        INTEGER*4 birthday(2),SYS$ASCTIM,trans1
        CHARACTER employee_id*5,last_name*14,first_name*10,middle_initial
        CHARACTER city*20,state*2,postal_code*5,ascii_date*23
        CHARACTER*8 data_base_key
        CHARACTER*25 address_data_1,address_data_2
```

(continued on next page)

Example 15–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE FORTRAN

```
C-----------------------------------------------------------
C Because the database was invoked in the main program
C with GLOBAL attributes, refer to it here as EXTERNAL.
C-----------------------------------------------------------

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH


C-------------------------------------------------------------
C The transaction was started in the DELETE_RECORD subroutine,
C so there is no need to start a transaction here.  Use the
C transaction handle to identify this request with the
C transaction started in DELETE_RECORD.  Use the dbkey found
C in DELETE_RECORD to locate the correct employee record.
C--------------------------------------------------------------

&RDB&   FOR (TRANSACTION_HANDLE trans1) E IN EMPLOYEES WITH
&RDB&           E.RDB$DB_KEY = data_base_key
&RDB&           GET
&RDB&                   employee_id = E.EMPLOYEE_ID;
&RDB&                   last_name = E.LAST_NAME;
&RDB&                   first_name = E.FIRST_NAME;
&RDB&                   middle_initial = E.MIDDLE_INITIAL;
&RDB&                   address_data_1 = E.ADDRESS_DATA_1;
&RDB&                   address_data_2 = E.ADDRESS_DATA_2;
&RDB&                   city = E.CITY;
&RDB&                   state = E.STATE;
&RDB&                   postal_code = E.POSTAL_CODE;
&RDB&                   birthday = E.BIRTHDAY;
&RDB&           END_GET
&RDB&   END_FOR

C-------------------------------------------------
C Display the EMPLOYEES record.  Use SYS$ASCTIM
C to convert the date stored in the database in
C binary to ASCII format.
C-------------------------------------------------

        CALL SYS$ASCTIM (, ascii_date, birthday, )
        TYPE 12000, employee_id, last_name, first_name,
        1               middle_initial, address_data_1,
        1               address_data_2, city, state,
        1               postal_code, ascii_date
12000   FORMAT (//   ' Employee_id:    ',A/
        1               ' Last name:      ',A/
        1               ' First name:     ',A/
        1               ' Middle initial: ',A/
        1               ' Address:        ',A,' ',A/
        1               ' City:         ',A/
        1               ' State:          ',A/
        1               ' Postal code:    ',A/
        1               ' Birthday:       ',A//)
```

(continued on next page)

Example 15–17 (Cont.) Using Data Manipulation Statements in Structured Programming in RDBPRE FORTRAN

```
C-----------------------------------------------------
C Return program control to the DELETE_RECORD subroutine.
C-----------------------------------------------------

        RETURN
        END
```

15.4.1 Using Handles in Structured Programming

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions
- Requests

Information on when and how to use request handles is supplied in Chapter 9. Section 15.4.2 and Section 15.4.4 discuss how to declare handles in an RDBPRE FORTRAN program.

15.4.2 Declaring and Initializing Handles

With the exception of the database handle, declaring handles in RDBPRE FORTRAN is similar to declaring any other program variable. The declaration and initialization of a database handle is done simply by specifying the handle in the DATABASE statement. You do not declare a database handle in the data declaration portion of your FORTRAN program. RDBPRE initializes the handle for you. You should not assign a value to a database handle with an assignment statement (or any other way).

User-specified request and transaction handles must be declared in the data declaration portion of your program. In FORTRAN, declare user-specified request and transaction handles as longwords (INTEGER*4) and initialize them to zero.

If you want to release the resources associated with a request handle, you can do so by issuing a FINISH statement, or, if you do not want to detach from the database, you can release the request by issuing a call to the RDB$RELEASE_REQUEST procedure with the following statement (where req1 is a user-supplied request handle):

```
status = (RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, req1))

IF ((status .AND. 1) .EQ. 0) THEN
   CALL SYS$PUTMSG(%REF(RDB$MESSAGE_VECTOR))
END IF
```

Declare the variable that holds the return status value as INTEGER*4.

Declare RDB$RELEASE_REQUEST as:

```
INTEGER*4 RDB$RELEASE_REQUEST
EXTERNAL  RDB$RELEASE_REQUEST
```

15.4.3 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

15.4.4 Declaring and Initializing Distributed Transaction Identifiers

Declaring distributed transaction identifiers in RDBPRE FORTRAN is similar to declaring any other program variable. Distributed transaction identifiers must be declared in the data declaration portion of your FORTRAN program. Declare a distributed transaction identifier as two longwords and initialize it to zero. You should not assign a value to a distributed transaction identifier with an assignment statement.

15.5 Using Callable RDO

The RDBPRE preprocessor statements do not include data definition statements. If you want to perform data definition within your preprocessed program, you must use the Callable RDO program interface. For example, during a batch process, or when others are not using the database, your program may define a temporary index on a field to facilitate Rdb/VMS performance during your program execution.

You can also use Callable RDO when your program needs the ability to form dynamic queries. That is, when your program will not know what a query is until run time. Otherwise, you should use the RDBPRE preprocessor when possible for all FORTRAN data manipulation operations. Preprocessed Rdb/VMS statements execute significantly faster than calls using the function RDB$INTERPRET.

When using Callable RDO, your program communicates with Rdb/VMS using the RDB$INTERPRET function. You call RDB$INTERPRET to pass your data manipulation or data definition statement to Rdb/VMS. Declare RDB$INTERPRET as an external integer (longword) function. The RDB$INTERPRET function returns a status value that indicates the success or failure of the function. The return status value is a systemwide condition value that indicates either success or a unique Rdb/VMS symbolic error code. Your program declares a longword variable to hold the return status value so you can test the success or failure of the call. (Refer to Chapter 10 and Section 15.6 in this chapter for further information on handling Rdb/VMS run-time exception conditions.)

The FORTRAN format of the RDB$INTERPRET calling sequence is:

```
ret-stat = RDB$INTERPRET('rdb-statement'[, [%DESCR(] host-var [)]],...])
```

The arguments for the RDB$INTERPRET function are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement

  The Rdb/VMS statement you pass to Rdb/VMS. Handle rdb-statement according to your language's rules for handling string literals or string variables.

- host-var

  A host language variable you pass to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*. You must include a by-descriptor passing mechanism when your language's default passing mechanism for the host language variable data type is not by descriptor. Refer to your FORTRAN language reference manual for the specific format of the passing mechanism.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some Rdb/VMS statements may produce unwieldy code in the call to the RDB$INTERPRET function. Instead, assign the Rdb/VMS statement string literal to a string variable. Then pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your Rdb/VMS data manipulation statements from the mechanics of using the RDB$INTERPRET function.

Callable RDO program development is explained in detail in Chapter 19.

The following section discusses the use of the INVOKE DATABASE statement and the scope of transactions in preprocessed programs that use Callable RDO.

15.5.1 Using the DATABASE Statement with Embedded Callable RDO

You must use an INVOKE DATABASE statement in your preprocessed RDBPRE program and a separate RDO INVOKE DATABASE statement in the embedded Callable RDO statements. To ensure that the preprocessor invokes the identical database for the preprocessed and the Callable RDO portions of the program, use the same database handle in each INVOKE DATABASE statement. Invoke the database:

- In the preprocessed program using a GLOBAL or EXTERNAL database handle.
- In the Callable RDO program by passing the database handle to the RDB$INTERPRET function.

For more information on database handles, see the section on handles in Chapter 9.

In Callable RDO, you must pass the database handle to RDB$INTERPRET as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both preprocessed and Callable RDO INVOKE DATABASE statements in the same program module. The preprocessor ignores any statement that is not preceded by the Rdb/VMS statement flag (&RDB&). You may also call a function or subroutine to perform the data definition with Callable RDO. In that case, use a preprocessed INVOKE DATABASE statement in the main module and the Callable RDO INVOKE DATABASE statement in the submodule.

For example, in the sample program for FORTRAN, the database is invoked with the GLOBAL attribute in the main program:

```
&RDB& DATABASE GLOBAL pers = FILENAME 'MF_PERSONNEL' DBKEY SCOPE IS FINISH
```

This program calls the subroutine named F_CALLABLE.FOR. The F_CALLABLE.FOR subroutine invokes the database using the RDB$INTERPRET function:

```
rdb_invoke = 'DATABASE !VAL = FILENAME MF_PERSONNEL'
STATUS = RDB$INTERPRET (rdb_invoke, %DESCR(db_handle))
IF ((STATUS .AND. 1) .NE. 0) THEN
        CALL callable_error_handler(STATUS)
        success = .FALSE.
END IF
```

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the Callable RDO sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

15.5.2 Embedding Data Definition Statements Using Callable RDO

Data definition statements require a read/write transaction. When an Rdb/VMS program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You must perform Callable RDO data definition statements within a read/write transaction. However, if you start a read/write transaction in the Callable RDO portion of your program, make sure that you commit or roll back any active transactions you started in the preprocessed portion of your program first. If a transaction is active in your program when you issue the START_TRANSACTION statement with a Callable RDO statement, your Callable RDO statement will return a run-time RDO error.

If you call the RDB$INTERPRET function for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view in Callable RDO and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist, and the preprocessor generates errors for those statements that refer to either the field, relation, or view. You can define indexes and constraints and any other database elements that are not referred to in the preprocessed code.

You can perform any preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than relying on implicit START_TRANSACTION declarations.

Example 15–18, from the F_DDL_STMNT.FOR subroutine, shows how to perform data definition tasks in RDBPRE FORTRAN programs.

Example 15–18 Embedding Data Definition Statements in RDBPRE FORTRAN

```
        SUBROUTINE ddl_stmnt

C-------------------------------------------------------------
C This subroutine demonstrates how to perform
C data definition tasks from an RDBPRE FORTRAN program.
C You must use the Callable RDO function, RDB$INTERPRET,
C to perform data definition tasks in preprocessed programs.
C--------------------------------------------------------------

        IMPLICIT NONE
        LOGICAL success
        INTEGER retry_count
        INTEGER*4 STATUS,RDB$INTERPRET,db_handle
        CHARACTER confirm,rdb_invoke*50,rdb_start*50,ddl_statement*256

C---------------------------------------------------------
C Invoke the database to make it known to Callable RDO.
C---------------------------------------------------------

        rdb_invoke = 'DATABASE !VAL = FILENAME MF_PERSONNEL'
        STATUS = RDB$INTERPRET (rdb_invoke, %DESCR(db_handle))
        IF ((STATUS .AND. 1) .NE. 0) THEN
                CALL callable_error_handler(STATUS)
                success = .FALSE.
        END IF

        WRITE (6,90)
90      FORMAT ('1',T25,'**** EXECUTE DDL ****'///)

C---------------------------------------------------------------------
C Prompt user for input.  Ordinarily, it would not be likely
C that you would ask a user to define an index for the database.
C This example serves only to show you how this type of task could be
C done within a FORTRAN environment.
C---------------------------------------------------------------------

        PRINT *,' Please enter the data definition statement to define'
        PRINT *,' or delete a temporary index, or type exit: '
        PRINT *,'  '
        PRINT *,' For example, to define an index for EMPLOYEES based'
        PRINT *,' on EMPLOYEE_ID, you might enter: '
        PRINT *,'  '
        PRINT *,'  define index emp_employee_id for employees duplicates
        1 are allowed.'
        PRINT *,'    employee_id.'
        PRINT *,'  end emp_employee_id index.'
        PRINT *,'  '
        PRINT *,' To delete this index, you might enter: '
        PRINT *,'  '
        PRINT *,'  delete index emp_employee_id.'
        PRINT *,'  '
        ACCEPT 130, ddl_statement
130     FORMAT (A256)
```

(continued on next page)

Example 15–18 (Cont.) Embedding Data Definition Statements in RDBPRE FORTRAN

```
        DO WHILE ((ddl_statement.NE.'EXIT ').AND.(ddl_statement.NE.'exit '))
            confirm = 'N'

            DO WHILE (confirm .EQ. 'N')
                PRINT *, '  '
                TYPE 1000
1000            FORMAT ('$',' Have you entered all data correctly? (Y/N): ')
                ACCEPT 1010, confirm
1010            FORMAT (A)
            END DO

                success = .FALSE.
                retry_count = 0


C--------------------------------------------------------
C Start a READ_WRITE transaction.
C--------------------------------------------------------

            DO WHILE ((retry_count .LT. 5) .AND. (.NOT. (success)))
                success = .TRUE.
                rdb_start = 'START_TRANSACTION READ_WRITE'
                STATUS = RDB$INTERPRET(%DESCR(RDB_START))
                IF ((STATUS .AND. 1) .NE. 0) THEN
                        success = .FALSE.
                        retry_count = retry_count + 1
                        CALL callable_error_handler(STATUS)
                END IF
            END DO

                success = .FALSE.
                retry_count = 0

C----------------------------------------------------
C Pass the data definition statement specified by the
C user to RDB$INTERPRET.
C----------------------------------------------------

            DO WHILE ((retry_count .LT. 5) .AND. (.NOT. (success)))
                success = .TRUE.
                STATUS = RDB$INTERPRET(%DESCR(ddl_statement))
                IF ((STATUS .AND. 1) .NE. 0) THEN
                        success = .FALSE.
                        retry_count = retry_count + 1
                        CALL callable_error_handler(STATUS)
                END IF
            END DO
```

(continued on next page)

Example 15–18 (Cont.) Embedding Data Definition Statements in RDBPRE FORTRAN

```
C--------------------------------------------------------------
C Inform user of success or failure of data definition task.
C--------------------------------------------------------------

            IF (success) THEN
                        PRINT *,' Transaction Successful'
                        CALL RDB$INTERPRET(%DESCR('COMMIT'))
            ELSE
                        PRINT *,' Transaction failed'
                        CALL RDB$INTERPRET(%DESCR('ROLLBACK'))
            END IF

C--------------------------------------------------------------------
C Ask user if he or she wants to define or delete another index.
C--------------------------------------------------------------------

            PRINT *,'   '
            PRINT *,' Please enter the data definition statement to define'
            PRINT *,' or delete a temporary index, or type exit: '
            PRINT *,'  '
            PRINT *,' For example, to define an index for EMPLOYEES based'
            PRINT *,' on EMPLOYEE_ID, you might enter: '
            PRINT *,'  '
            PRINT *,'  define index emp_employee_id for employees duplicates
        1 are allowed.'
            PRINT *,'    employee_id.'
            PRINT *,'  end emp_employee_id index.'
            PRINT *,'  '
            PRINT *,' To delete this index, you might enter: '
            PRINT *,'  '
            PRINT *,'  delete index emp_employee_id.'
            PRINT *,'  '
            ACCEPT 130, ddl_statement

        END DO

        RETURN
        END
```

15.6 Handling Rdb/VMS Run-Time Errors

Before reading this section, you should be familiar with the information contained in Chapter 10 of this manual. Chapter 10 discusses error handling concepts; this section contains information that, for the most part, is specific to error handling in RDBPRE FORTRAN.

This section describes how to detect Rdb/VMS errors that occur at run time, how to display the accompanying messages, and how to recover from errors. In most cases, this section assumes that you have debugged the program for errors in both Rdb/VMS and host language statements. This section discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your FORTRAN user’s guide for such information.

If you choose to combine Callable RDO and RDBPRE DML statements, use separate error handling routines for each one. See Chapter 19 for information on handling Callable RDO errors.

15.6.1 Error Handling

RDBPRE FORTRAN enables you to detect errors with the ON ERROR clause. If an error occurs in an Rdb/VMS data manipulation statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- The ON ERROR clause
- Determining which error has occurred using the LIB$MATCH_COND run-time library routine
- Error message display using the SYS$GETMSG, SYS$PUTMSG and LIB$SIGNAL routines

Information on creating user-supplied error messages is contained in Chapter 10.

15.6.2 Detecting Errors Using the ON ERROR Clause

You can use the ON ERROR clause only in preprocessed programs. All data manipulation statements except INVOKE DATABASE and DECLARE_STREAM offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you can include one or more host language or Rdb/VMS statements, or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

Note *Do not use the START_TRANSACTION statement within the ON ERROR . . . END_ERROR block.*

If you do not use the ON ERROR clause and an Rdb/VMS error occurs, Rdb/VMS passes the error to the VMS run-time library routine, LIB$STOP, which sets the severity level to 4 (FATAL) and forces program termination.

See Chapter 10 for a more complete description of the ON ERROR clause.

The FORTRAN code fragment in the example that follows shows the placement of the ON ERROR clause and host language statements within a MODIFY operation.

```
&RDB&                       FOR E IN EMPLOYEES WITH
&RDB&                               E.EMPLOYEE_ID = employee_id
&RDB&                           MODIFY E USING
&RDB&                           ON ERROR
                                    success = .FALSE.
                                    CALL error_handler(RDB$STATUS,success)
                                    IF (success) THEN
                                            retry_count = 5
                                    END IF
&RDB&                           END_ERROR
&RDB&                               E.ADDRESS_DATA_1 = address_data_1;
&RDB&                               E.ADDRESS_DATA_2 = address_data_2;
&RDB&                               E.CITY = city;
&RDB&                               E.STATE = state;
&RDB&                               E.POSTAL_CODE = postal_code;
&RDB&                           END_MODIFY
&RDB&                       END_FOR
```

15.6.3 Determining Which Errors Have Occurred

After detecting an error, you want to determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation. You determine which error has occurred by evaluating the symbolic value of the error code.

15.6.3.1 Using Symbolic Error Codes All communication with Rdb/VMS is done through procedure calls. In preprocessed programs, the preprocessor converts Rdb/VMS statements to host language calls to Rdb/VMS procedures. Every procedure returns a status value into a program variable, RDB$STATUS, that is declared by the preprocessor. The return status value is a longword that identifies a unique message in the system message file. The return status value may indicate success, in which case data manipulation continues uninterrupted. Or this value may signal an error, in which case control passes to the error handler.

In RDBPRE FORTRAN programs, the preprocessor names this variable RDB$STATUS and declares it to be a longword. The return status value is the second element of a 20-longword array, RDB$MESSAGE_VECTOR. (The RDB$MESSAGE_VECTOR array is the message vector that Rdb/VMS uses to pass information to and from FORTRAN programs.)

Each error generated by an RDBPRE statement is represented as a symbolic error code. You can use these symbolic error codes to control program logic for specific errors. When the Rdb/VMS ON ERROR clause detects an error, your error handler should:

- Evaluate the symbolic error code either by calling the LIB$MATCH_COND routine or by using a FORTRAN equality test

- Direct program logic with one or more FORTRAN host language statements, such as the Block IF, Arithmetic IF, Computed GO TO, or Assigned GO TO statements

Although symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. Chapter 10 explains why this is recommended.

15.6.3.2 Declaring Symbolic Error Codes Rdb/VMS symbolic error codes are longword values. In FORTRAN programs, they should be declared separately as external variables and as integer longword variables. For example:

```
INTEGER*4 RDB$_LOCK_CONFLICT
INTEGER*4 RDB$_DEADLOCK

EXTERNAL RDB$_LOCK_CONFLICT
EXTERNAL RDB$_DEADLOCK
```

15.6.3.3 Calling LIB$MATCH_COND When you want to determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine, LIB$MATCH_COND.

You also can evaluate the return status value directly with host language statement or statements, without calling the LIB$MATCH_COND routine. Generally, host language statements will use fewer resources than LIB$MATCH_COND. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. You must then link your program again under the new version so that the program will detect the correct error codes. The LIB$MATCH_COND routine matches only the condition ID of the return status value and is unaffected by changes in severity levels or facility names.

The LIB$MATCH_COND routine compares the first parameter to each of the remaining parameters in its parameter list. If a match is found, it returns the position in the parameter list of the matching parameter. If no match is found, the LIB$MATCH_COND routine returns a zero. You should pass the return status value to the LIB$MATCH_COND routine as the first parameter in the parameter list. In the remaining part of the parameter list, pass the error codes you wish to compare to the return status value. If one of these error codes matches the return status value, the LIB$MATCH_COND routine returns the position of the matching parameter in the parameter list.

For example, suppose you want to determine if RDB$STREAM_EOF, RDB$_DEADLOCK, or RDB$NOT_VALID is the return status value. Pass to the LIB$MATCH_COND routine the parameter list that contains RDB$STATUS, RDB$_STREAM_EOF, RDB$_DEADLOCK, and RDB$_NOT_VALID. If RDB$STATUS equals RDB$_DEADLOCK, then the LIB$MATCH_COND routine returns a value of 2 because RDB$_DEADLOCK is the second parameter in the parameter list.

Next, use the value that the LIB$MATCH_COND routine returns to determine the path of your error handler's conditional statement. To continue our example, assume you use a computed GO TO statement as the error handler's conditional statement. In this example, your computed GO TO statement evaluates the value returned by the LIB$MATCH_COND routine, and your program falls through to the second label of the GO TO statement. Your program performs the statement(s) associated with the label statement. These statements might print a message to the terminal, roll back the transaction, and return program control to a point before the transaction was opened. Or they might call a more complex routine to perform these and other actions.

The FORTRAN format of the call to the LIB$MATCH_COND routine is:

```
err-match = LIB$MATCH_COND([%REF(]ret-stat[)],%LOC(symb_name)[,...])
```

The arguments for this FORTRAN call are:

- err-match

  A numeric variable that holds the integer that identifies the symbol matched.

- ret-stat

  A program variable (RDB$STATUS) that holds the return status value of the last call to the database.

- symb-name

  One or more symbolic error codes, (or the variable names you have assigned to them) that you want to match against ret-stat. The symbolic error codes are longwords and are passed by reference.

Declare the LIB$MATCH_COND routine as EXTERNAL in FORTRAN programs.

Example 15–19 demonstrates the use of the LIB$MATCH_COND routine in a FORTRAN error handling routine. This error handler could be called from another program that:

- Detects errors with an ON ERROR clause
- Includes a statement within the ON ERROR . . . END_ERROR block that sets the value of a success flag to FALSE when the ON ERROR clause is executed

This error handling routine:

- Receives the return status and the success flag values
- Opens a file to record the error messages
- Uses the LIB$MATCH_COND routine to determine which error has occurred

- Uses a computed GO TO statement to take different actions depending on which error has occurred
- Sets the success flag to true if corrective error handling could take place
- Closes the file that records the error messages

Example 15–19 Using LIB$MATCH_COND in RDBPRE FORTRAN

```
        SUBROUTINE error_handler(RDB$STATUS,success)

C------------------------------------------------------------
C This subroutine handles run-time errors identified by
C the ON ERROR clause in the sample FORTRAN programs.
C------------------------------------------------------------

        IMPLICIT NONE
        LOGICAL success

C-----------------------------------------------------
C Declare variables and symbolic error codes and system
C service library routines.
C-----------------------------------------------------

        CHARACTER*80 msg_txt
        INTEGER*4 RDB$_LOCK_CONFLICT,RDB$_DEADLOCK,RDB$_NO_DUP
        INTEGER*4 RDB$_NOT_VALID,RDB$_INTEG_FAIL,RDB$_NO_RECORD
        INTEGER*4 RDB$STATUS,LIB$CALLG,SYS$GETMSG,LIB$SIGNAL
        INTEGER*4 LIB$MATCH_COND,SYS$PUTMSG,error_match
        EXTERNAL RDB$_LOCK_CONFLICT,RDB$_DEADLOCK,RDB$_NO_DUP
        EXTERNAL RDB$_NOT_VALID,RDB$_INTEG_FAIL,RDB$_NO_RECORD
        EXTERNAL LIB$MATCH_COND

&RDB&   DATABASE EXTERNAL pers=FILENAME 'MF_PERSONNEL'
&RDB&   DBKEY SCOPE IS FINISH

        OPEN (UNIT=3, FILE='error_file.log', STATUS='new')

C-----------------------------------------------------
C Use LIB$MATCH_COND to determine which of a series
C of errors might have occurred.
C-----------------------------------------------------

        error_match = LIB$MATCH_COND(%REF(RDB$STATUS),
        1               %LOC(RDB$_LOCK_CONFLICT),
        1               %LOC(RDB$_DEADLOCK),
        1               %LOC(RDB$_NO_DUP),
        1               %LOC(RDB$_NOT_VALID),
        1               %LOC(RDB$_INTEG_FAIL),
        1               %LOC(RDB$_NO_RECORD))
```

(continued on next page)

Example 15–19 (Cont.) Using LIB$MATCH_COND in RDBPRE FORTRAN

```
C---------------------------------------------------
C The GO TO statement directs program to appropriate
C statements to execute depending on the error
C that was identified.
C---------------------------------------------------

        GO TO (10,10,20,30,40,50) error_match

C       Unexpected error
        WRITE (5,90)
        WRITE (3,90)
        CALL SYS$GETMSG(%VAL(Rdb$STATUS),,%DESCR(msg_txt))
        WRITE (5,95) msg_txt
        WRITE (3,95) msg_txt
        CALL LIB$CALLG(%REF(Rdb$MESSAGE_VECTOR),
        1               %VAL(LIB$SIGNAL))
        RETURN

C       Lock Conflict and deadlock
10      CALL SYS$PUTMSG(%REF(Rdb$MESSAGE_VECTOR))
        WRITE (5,100)
        WRITE (3,100)
        RETURN

C       No duplicates allowed
20      CALL SYS$PUTMSG(%REF(Rdb$MESSAGE_VECTOR))
        WRITE (5,200)
        WRITE (3,200)
        success = .TRUE.
        RETURN

C       Invalid data
30      CALL SYS$PUTMSG(%REF(Rdb$MESSAGE_VECTOR))
        WRITE (5,300)
        WRITE (3,300)
        success = .TRUE.
        RETURN

C       Integrity failure
40      CALL SYS$PUTMSG(%REF(Rdb$MESSAGE_VECTOR))
        WRITE (5,400)
        WRITE (3,400)
        success = .TRUE.
        RETURN

C       Record deleted
50      WRITE (5,500)
        WRITE (3,500)
        success = .TRUE.
        RETURN
```

(continued on next page)

Example 15–19 (Cont.) Using LIB$MATCH_COND in RDBPRE FORTRAN

```
90      FORMAT (' ',' Unexpected error - terminating program'/)

95      FORMAT (' ',A80)

100     FORMAT (' ',' Another user is accessing data you
        1attempted to access',/,' Please choose a new value
        1and try again'/)

200     FORMAT (' ',' You attempted to insert a record with a
        1value already on file'/)

300     FORMAT (' ',' In the data you entered, you specified
        1 an invalid value',/,' Please correct the error and
        1try again')

400     FORMAT (' ',' In the data you entered, you violated
        1a constraint',/,' Please correct the error and try
        1again'/)

500     FORMAT (' ',' Record entered has already been deleted'/)

        END
```

15.6.4 Displaying Error Messages

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and terminate your program, you can call the LIB$SIGNAL routine
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code

Information on creating user-supplied error messages is contained in Chapter 10.

15.6.4.1 Calling LIB$SIGNAL Call the LIB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and either continue program execution, or terminate program execution. LIB$SIGNAL is a VMS Run-Time Library routine that:

- Receives the signal argument list from the signaling procedure

  This list is made up of the return status value and a set of optional arguments that provide information to condition handlers.

- Copies this signal argument list and uses it to create a signal argument vector

  The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.

- Causes a signal condition which causes the appropriate catchall condition handler to pass the signal argument vector to the SYS$PUTMSG system service

  The SYS$PUTMSG system service calls SYS$GETMSG to retrieve the message from the error messages file and then formats and displays the error message on your terminal.

- Resignals the error

  If the error is not fatal, program execution continues. If the error is fatal, the host language error handler signals the error to the VMS default condition handler, which terminates program execution.

In FORTRAN, you can continue program execution after the call to the LIB$SIGNAL routine even when the error is fatal. See Section 15.6.5 for information on how to continue program execution after a call to LIB$SIGNAL.

15.6.4.2 Methods of Calling LIB$SIGNAL The recommended method of calling LIB$SIGNAL in RDBPRE programs is to pass the message vector, RDB$MESSAGE_VECTOR, and the LIB$SIGNAL routine to the run-time library function, LIB$CALLG.

This method ensures that any FAO arguments that exist in the message vector will be formatted correctly. In addition, this method ensures that any additional error messages that clarify the nature of the program error will be returned to your program. For these reasons, Digital recommends that you always call LIB$SIGNAL with LIB$CALLG.

You can also pass the return status value, RDB$STATUS, to the LIB$SIGNAL routine. However, this method is not recommended. If you pass RDB$STATUS to the LIB$SIGNAL routine and FAO arguments exist in the Rdb/VMS error message, LIB$SIGNAL may be unable to format the Rdb/VMS error message correctly. In this case, your program may terminate abruptly or may provide incompletely formatted error messages.

If your application requires that you call LIB$SIGNAL without LIB$CALLG, be certain that the error message does not contain FAO arguments. Figure 10–1 in Chapter 10 illustrates the format of the message vector.

15.6.4.3 The Format of the LIB$SIGNAL Calling Sequence with RDB$MESSAGE_VECTOR and RDB$STATUS

The FORTRAN format of the LIB$SIGNAL calling sequence with the message vector (RDB$MESSAGE_VECTOR) is:

```
CALL LIB$CALLG([%REF(]RDB$MESSAGE_VECTOR[)],[%VAL(]LIB$SIGNAL[)])
```

The LIB$SIGNAL argument is the run-time library routine that will receive RDB$MESSAGE_VECTOR. This argument is passed by value in FORTRAN.

When using the LIB$CALLG routine to pass the message vector, you must declare LIB$CALLG as an external integer function and LIB$SIGNAL as either an external function or an intrinsic function in FORTRAN.

An earlier example, Example 15–19, demonstrates how to call LIB$SIGNAL with LIB$CALLG. The FORTRAN format of the LIB$SIGNAL calling sequence with the return status value is:

```
CALL LIB$SIGNAL([%VAL(]RDB$STATUS[)])
```

When using the LIB$SIGNAL routine to pass the return status value, you must declare LIB$SIGNAL as an external integer function.

15.6.4.4 Calling SYS$PUTMSG

Call the SYS$PUTMSG system service when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service displays the error message on the terminal and writes it to the error file designated by the logical name SYS$ERROR. You can define SYS$ERROR at the DCL level to be your program error file when you want the SYS$PUTMSG system service to write an Rdb/VMS error message to it.

The first parameter in the call to the SYS$PUTMSG system service is the message vector RDB$MESSAGE_VECTOR. Figure 10–1 in Chapter 10 illustrates the format of the message vector. The SYS$PUTMSG system service can accept other optional parameters that specify a routine that receives control during message processing, and the facility name to be used in displaying the message (if you want the facility to be different from the default facility prefix that is associated with the message). The message vector is required; you may omit the optional parameters. See the *VMS System Services Volume* for a complete description of the SYS$PUTMSG system service.

The FORTRAN format of the SYS$PUTMSG calling sequence is:

```
CALL SYS$PUTMSG([%REF(] RDB$MESSAGE_VECTOR[)])
```

Declare the SYS$PUTMSG system service as an external integer function. See an earlier example, Example 15–19, for a demonstration of the use of the SYS$PUTMSG system service.

15.6.4.5 Calling SYS$GETMSG Call the SYS$GETMSG system service when you want to use an error message message generated by Rdb/VMS within your program and continue program execution.

The first parameter in the call to the SYS$GETMSG system service is the Rdb/VMS return status value, the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program as the second parameter of the call. You must declare a string to receive the message. Your program can then manipulate this string in any way it chooses. Your program can:

- Display the string
- Write the string to a file

You can also evaluate character substrings within the string, but Digital recommends that you do not use this method. The message text may change from one version of Rdb/VMS to the next.

The SYS$GETMSG system service requires a parameter to receive the length of the message string. You may omit the actual parameter, but you must include a comma to signify the argument. The SYS$GETMSG system service accepts other optional parameters that define what is included in the returned message and receives the FAO count of the message. You may omit these parameters; if you do, all components of the message are returned. See the *VMS System Services Volume* for further information on the SYS$GETMSG system service.

The SYS$GETMSG system service does not format the FAO arguments in the error message; instead, it returns the error message with format parameters embedded in it. If your error message contains a view name, for example, SYS$GETMSG will return the message:

```
<View !AC can not be updated>
```

You can call the SYS$FAO system service to format the FAO arguments in the message SYS$GETMSG returns to your program. However, when the error message contains FAO arguments, it is preferable to call the SYS$PUTMSG system service rather than SYS$GETMSG. The optional parameters that you can specify with the SYS$GETMSG system service are not shown below. For more information on SYS$GETMSG, see the *VMS System Services Volume.*

The FORTRAN format of the SYS$GETMSG calling sequence is:

```
CALL SYS$GETMSG([%VAL(]RDB$STATUS[)],[%REF(msg-len)],[%DESCR(]msg-string[)])
```

The arguments of this calling sequence are:

- msg-len

  A word that holds the number of characters written into msg-string. This is not an optional parameter; if you omit it, you must use a comma. This is passed by reference.

- msg-string

  A string variable that holds the returned error message. The maximum length of any message that can be returned is 256 bytes. This is passed by descriptor.

Declare SYS$GETMSG as an external integer function in FORTRAN. See an earlier example, Example 15–19, for a demonstration of the use of the SYS$GETMSG system service.

15.6.5 Handling Fatal Errors

In many instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error with the LIB$MATCH_COND routine or host language statement or statements to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or SYS$GETMSG system services to output an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

In FORTRAN you can also call the LIB$SIGNAL routine to display the error message, but you must use the LIB$ESTABLISH routine to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary before calling the LIB$SIGNAL routine. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions
- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the LIB$SIGNAL routine without establishing a condition handler, LIB$SIGNAL displays the error message and terminates your program. Perform any cleanup before making the call to LIB$SIGNAL. However, if your cleanup includes any Rdb/VMS statements (such as ROLLBACK), these new calls to the database will change the return status value contained in RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in this case, the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling the LIB$SIGNAL routine without any cleanup operations is not recommended.

16

Using the RDML Program Environment

This chapter describes how to develop application programs that access an Rdb/VMS database using Relational Data Manipulation Language (RDML) preprocessed programs. The chapter presents the following topics:

- Differences in syntax between RDO and RDML
- Using the DECLARE_VARIABLE clause to declare host language variables
- Copying VAX CDD/Plus definitions to declare host language variables

Most of the information you need to develop an RDML program is contained in Chapter 9, Chapter 17, and Chapter 18. This chapter provides you with information that is specific to RDML and applies to both of the RDML programming languages, C and Pascal.

16.1 RDML Program Development

To ensure effective program development, you should:

- Develop your queries in RDO

  You need to know which databases, relations, and fields your program accesses. Special characteristics of the relations, views, and field definitions in the database will determine the most efficient form for a query.

- Determine host language variables

  Your program usually needs to declare host language variables that pass values to and accept values from the database. You need to be aware of the existing data types, data restrictions, input constraints, and trigger definitions that are part of the design of the databases you access.

- Convert your query to the program environment

  In a typical application, the database is the source of records for reports and calculations, as well as the target for updates. The host language provides logic for operations such as flow control, error handling, conditional processing, numeric manipulation, and input/output.

16.1.1 Differences in RDO and RDML Syntax

The RDML data manipulation statements are similar to RDO statements. With these statements you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. Refer to the *RDML Reference Manual* for a complete description of the RDML data manipulation statements.

The syntax you use for RDML statements is not identical to the statement syntax you use in RDO. When you incorporate tested RDO statements into an RDML program, you need to consider these areas:

- Using a host language display statement instead of the PRINT statement
- Placing field data into host language variables
- Nesting a FETCH operation within a host language loop
- Using the FOR with segmented strings statement instead of the START_SEGMENTED_STRING statement
- Using the STORE segmented string statement instead of the CREATE_SEGMENTED_STRING statement
- Using the semicolon, the Pascal statement separator, and C statement terminator
- Using the ON ERROR and AT END clauses to detect Rdb/VMS errors

Examples of how to use these statements are contained in Chapter 17 and Chapter 18.

16.1.2 Declaring Host Language Variables

An easy way to declare host language variables in RDML is to use the RDML DECLARE_VARIABLE clause. The DECLARE_VARIABLE clause lets you declare a host language variable by referring to a database field. The variable inherits the data type and size attributes associated with the field. You can use the DECLARE_VARIABLE clause to declare data types for most fields. The exceptions are described in Chapter 17 and Chapter 18.

The RDML BASED ON clause extracts the data type and size of fields from the database and allows you to declare Pascal TYPE(s) and C typedef(s) based on these fields. When you preprocess your program, the RDML preprocessor assigns the data type and size attributes associated with the field to the variable or function you declare using the BASED ON clause. The data types that the BASED ON clause generates for a variable (when it is different from the data type of the database field) are discussed in Chapter 17 and Chapter 18.

Another way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus. You can copy field and relation definitions (which include all the fields within the relation). However, you must be careful to copy only those relation and field definitions with data types that are supported by your host language. If an Rdb/VMS data type is flagged with a dagger in Table 8–5 or Table 8–8, you should check the conversion performed by the data dictionary and make sure that the data type is appropriate for your application.

You can copy data definitions into your program from the data dictionary if you are using VAX C V2.4 or higher, or VAX Pascal V3.6 or higher. However, copying from the data dictionary is not a completely automatic process for you. Be careful to avoid the following:

- Naming conflicts

  You must ensure that relation and field names copied into your program do not conflict with the Pascal or C naming rules and are not Pascal or C reserved words. If there are any naming conflicts, you must change the name in the appropriate database definition before copying the data dictionary definition. For information about changing database attribute definitions, see the *VAX Rdb/VMS RDO and RMU Reference Manual*.

- Field names that are not unique

  Field definitions copied from the data dictionary are likely to contain names that are not unique. Be sure to qualify any non-unique field name by the relation name that contains it. If you do not qualify field names that are not unique, you will get compile-time errors indicating ambiguous reference. You can use the appropriate compile qualifier to print out the copied definition or definitions into your listing (LIS) file and then check for field names that are not unique.

- Data type conflicts

  The RDML preprocessor translates copied data dictionary data types into equivalent host language data types where possible. The Pascal and C compilers, however, do not perform the data type conversions that the RDML preprocessor performs. If an RDML data type is flagged with a dagger (†) in Table 8–5 or Table 8–8 as “unsupported” by your host language, you cannot copy that definition into your program. For example, you cannot use the %DICTIONARY statement to extract the data dictionary definition for a field that is a SCALED INTEGER data type. RDML/Pascal uses different host language data types for scaled integers than does the Pascal %DICTIONARY statement. Pascal ignores the scaled attribute. However, RDML uses a data type that can still represent the values stored in the database. For this reason, you should use the DECLARE_VARIABLE clause to declare a variable for a field that is a SCALED INTEGER data type.

If you must copy the definition, change the affected Rdb/VMS field definition to a data type that your host language does support.

Furthermore, when you use the #dictionary control line in VAX C to declare a variable for a TEXT field, be aware that you cannot use a *strcpy* or similar function to copy strings into the variable. The variable declared by the #dictionary control line does not leave space for the null terminator that is conventionally used to terminate strings in C programs. However, if you use the RDML DECLARE_VARIABLE clause instead of the #dictionary control line you do not need to be concerned with this issue. The DECLARE_VARIABLE clause declares the variable for a TEXT field with an extra space provided for the null terminator.

You can copy field and relation definitions (and the definitions for the fields contained within the relation) from the data dictionary. Relation definitions are stored in the data dictionary as objects under the RDB$RELATIONS directory. Field definitions are stored in the data dictionary as objects under the RDB$FIELDS directory. To copy a relation into your program, specify the location of your dictionary database, then specify the database and the dictionary path name of the relation that you want to copy.

For example, to copy the EMPLOYEES relation from an MF_PERSONNEL database, specify the following in your host language statement to copy from the dictionary:

```
DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES
```

The data dictionary is organized as a hierarchy of directories and objects. You can use the Common Dictionary Operator utility (CDO) to display on your terminal all the data dictionary entities for a particular database. By observing the display, you can identify the data dictionary path name you need to include in the copy statement. For additional information about the data dictionary structure, see the *VAX Common Data Dictionary Utilities Reference Manual*.

To list field definitions on your terminal, you must first use the CDO ENTER command to assign CDO directory names to record definitions within a CDD$DATABASE definition. Then enter the CDO SHOW ALL/FULL command to see the field definitions for all the records that you have assigned CDO directory names with the ENTER command.

For example, if you want to see the field definitions for the EMPLOYEES relation in the MF_PERSONNEL database, enter the following commands:

```
$ DICTIONARY OPERATOR
CDO> ENTER RECORD EMPLOYEES FROM DATABASE MF_PERSONNEL
CDO> SHOW ALL/FULL
Definition of record EMPLOYEES
|   Contains field          EMPLOYEE_ID
|   |   Based on                ID_NUMBER
|   |   |   Description             ' Generic employee ID '
|   |   |   Datatype                text size is 5 characters
|   Contains field          LAST_NAME
|   |   Description             ' Generic last name '
|   |   Datatype                text size is 14 characters
|   Contains field          FIRST_NAME
|   |   Description             ' Generic first name '
|   |   Datatype                text size is 10 characters
|   Contains field          MIDDLE_INITIAL
|   |   Description             ' Generic middle initial '
|   |   Datatype                text size is 1 characters
|   |   Missing_value           " "
|   |   DTR Edit_string         X.
|   Contains field          ADDRESS_DATA_1
|   |   Description             ' Street name '
|   |   Datatype                text size is 25 characters
|   |   Missing_value           "                         "
|   Contains field          ADDRESS_DATA_2
|   |   Description             ' Mail stops, suite addresses,
|   |                             street numbers . . .  '
|   |   Datatype                text size is 25 characters
|   |   Missing_value           "                         "
|   Contains field          CITY
|   |   Description             ' City name '
|   |   Datatype                text size is 20 characters
|   |   Missing_value           "                    "
|   Contains field          STATE
|   |   Description             ' State abbreviation (or DISTRICT) '
|   |   Datatype                text size is 2 characters
|   |   Missing_value           "  "
|   Contains field          POSTAL_CODE
|   |   Description             ' Postal code (in US = ZIP)'
|   |   Datatype                text size is 5 characters
|   |   Missing_value           "     "
|   Contains field          SEX
|   |   Description             ' M, F '
|   |   Datatype                text size is 1 characters
|   |   Missing_value           "?"
|   |   Valid if                (((SEX EQ "M") OR (SEX EQ "F"))
|   |                              OR (SEX MISSING ))
|   Contains field          BIRTHDAY
|   |   Based on                STANDARD_DATE
|   |   |   Description             ' Generic date field '
|   |   |   Datatype                date
|   |   |   Missing_value           17-NOV-1858 00:00:00.00
|   |   |   DTR Edit_string         DD-MMM-YYYY
|   Contains field          STATUS_CODE
|   |   Description             ' A number '
|   |   Datatype                text size is 1 characters
|   |   Missing_value           "N"
|   |   Valid if                ((((STATUS_CODE EQ "0") OR
                                   (STATUS_CODE EQ "1")) OR
                                   (STATUS_CODE EQ "2")) OR
                                   (STATUS_CODE MISSING ))
```

```
Definition of database  MF_PERSONNEL
|   database uses RDB database MF_PERSONNEL
|   database in file MF_PERSONNEL
|   |   fully qualified file DISK:[MYDATABASE]MF_PERSONNEL.RDB;
CDO>
```

16.1.3 The C #dictionary Control Line

The #dictionary control line is specific to C, and lets you extract data definitions and include these definitions in your program.

The format of the #dictionary control line is:

```
#dictionary dictionary-path-name
```

The *dictionary-path-name* argument specifies the full or relative data dictionary path name that identifies the location in the data dictionary of the object definition you want to copy. Enclose the path name in pairs of double quotation marks. You can use a logical name for dictionary-path-name.

For example, to extract the definition of the EMPLOYEES relation, place the following statement after the database statement, but before the main C function:

```
#dictionary "disk:[myfiles]mf_personnel.rdb$relations.employees"
```

If you specify the /LISTING qualifier and either the /SHOW=DICTIONARY or /SHOW=ALL qualifier in the compile command line, the translation of the data dictionary record description into C is included in the LIS file and marked with the letter D in the margin.

The following segment from a LIS file (edited slightly to improve readability) shows the C translated text of the EMPLOYEES relation definition. Note the data type conversions for the DATE data type of the BIRTHDAY field.

```
  457   /*    7 */   typedef
  458
  459   #dictionary "disk:[myfiles]mf_personnel.rdb$relations.employees"
        D       /* CDD Path Name is "disk:[my_files]mf_personnel.
                                     rdb$relations.employees" */
        D       struct employees
        D       {
        D           /*  Generic employee ID  */
        D           char employee_id [5];
           /* text */
        D           /*  Generic last name  */
        D           char last_name [14];
           /* text */
        D           /*  Generic first name  */
        D           char first_name [10];
           /* text */
        D           /*  Generic middle initial  */
        D           char middle_initial;
           /* text */
        D           /*  Street name  */
        D           char address_data_1 [25];
           /* text */
        D           /*  Mail stops, suite addresses, ...  */
        D           char address_data_2 [25];
           /* text */
        D           /*  City name  */
        D           char city [20];
           /* text */
        D           /*  State abbreviation (or DISTRICT)  */
        D           char state [2];
           /* text */
        D           /*  Postal code (in US = ZIP) */
        D           char postal_code [5];
           /* text */
        D           /*  M, F  */
        D           char sex;
           /* text */
        D           /*  Generic date field  */
        D           struct { char cc_cdd$_unsupported_#1 [8]; } birthday;
           /* absolute date and time */
        D           /*  A number  */
        D           char status_code;
           /* text */
        D       };
%CC-I-UNSUPPTYPE, The CDD description for "birthday"
                specifies a data type not supported in C.
```

See the chapter on preprocessor control lines in *Programming in VAX C* for more information.

16.1.4 The Pascal %DICTIONARY Statement

Pascal lets you use the data dictionary to define a TYPE record structure for database relations in your Pascal program. After identifying the data dictionary path name of the relation definition you want to copy, use the Pascal %DICTIONARY statement to create a record structure in the TYPE section of your program. Then use the record structure from the TYPE section of your program to declare host language variables in the VAR section. To copy multiple definitions, use the %DICTIONARY statement to define record structures as needed.

The format for the %DICTIONARY statement is:

```
TYPE
  %DICTIONARY 'dictionary-path-name'
```

The *dictionary-path-name* argument specifies the full or relative data dictionary path name that identifies the location in the data dictionary of the object definition you want to copy. Enclose the path name in pairs of single quotation marks. You can use a logical name for dictionary-path-name.

The following program segment shows you how to copy a dictionary definition into a Pascal program. Note that Pascal translates the dictionary definition into a record structure and subordinate fields. To allocate memory to the fields of the record, you must declare a host language variable in the VAR section using the record structure you declared in the TYPE section.

```
TYPE
(* copy employees relation from CDD/Plus *)

%DICTIONARY 'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'

VAR
  EMP  :  EMPLOYEES; (* declare employees record *)
 . . .                 .
```

You can obtain a compiled program LIS file that includes the translated definition using the /LISTING and /SHOW=DICTIONARY qualifiers in the compile command line. For example:

```
$ PASCAL PROG2/LIST/SHOW=DICTIONARY
```

The following segment from the LIS file shows the Pascal translated text of the EMPLOYEES relation definition. Note the data type conversions for the DATE data type of the BIRTHDAY field.

```
00479      0  0 (*    6 *)TYPE
00480      0  0 (*    7 *)%DICTIONARY
                          'DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES'
00481  DC  0  0 { CDD Path Name =>
                          DISK:[MYFILES]MF_PERSONNEL.RDB$RELATIONS.EMPLOYEES }
00482  D   0  0 EMPLOYEES =  PACKED RECORD
00483  DC  0  0      {  Generic employee ID  }
00484  D   0  0      EMPLOYEE_ID :  PACKED ARRAY [1..5] OF CHAR;
00485  DC  0  0      {  Generic last name  }
00486  D   0  0      LAST_NAME :  PACKED ARRAY [1..14] OF CHAR;
00487  DC  0  0      {  Generic first name  }
00488  D   0  0      FIRST_NAME :  PACKED ARRAY [1..10] OF CHAR;
00489  DC  0  0      {  Generic middle initial  }
00490  D   0  0      MIDDLE_INITIAL :  CHAR;
00491  DC  0  0      {  Street name  }
00492  D   0  0      ADDRESS_DATA_1 :  PACKED ARRAY [1..25] OF CHAR;
00493  DC  0  0      {  Mail stops, suite addresses, street numbers . . .  }
00494  D   0  0      ADDRESS_DATA_2 :  PACKED ARRAY [1..25] OF CHAR;
00495  DC  0  0      {  City name  }
00496  D   0  0      CITY :  PACKED ARRAY [1..20] OF CHAR;
00497  DC  0  0      {  State abbreviation (or DISTRICT)  }
00498  D   0  0      STATE :  PACKED ARRAY [1..2] OF CHAR;
00499  DC  0  0      {  Postal code (in US = ZIP) }
00500  D   0  0      POSTAL_CODE :  PACKED ARRAY [1..5] OF CHAR;
00501  DC  0  0      {  M, F  }
00502  D   0  0      SEX :  CHAR;
00503  DC  0  0      {  Generic date field  }
00504  D   0  0      BIRTHDAY : [BYTE(8)] RECORD END; { absolute date/time }
                     1
```

```
%Pascal-I-CDDUNSTYP, (1) Unsupported CDD datatype 'absolute date/time'
00505  DC  0  0      {  A number  }
00506  D   0  0      STATUS_CODE :  CHAR;
00507  D   0  0      END; { record EMPLOYEES }
00508   C  0  0 (*    8 *)
00509      0  0 (*    9 *)   var
00510      0  0 (*   10 *)     EMP:EMPLOYEES;
00511      0  1 (*   11 *)   begin
```

17

Using the RDML/C Program Environment

This chapter describes how to access an Rdb/VMS database using VAX C programs and the RDML preprocessor interface. This chapter presents the following main topics:

- Using Relational Data Manipulation Language (RDML) statements
- Using Rdb/VMS data definition statements
- Error handling in RDML/C

Most examples in this chapter are available on line. The Rdb/VMS installation procedure writes the sample programs to the directory identified by the logical name RDM$DEMO. The file names for these programs are C_SAMPLE.RC, C_CALL_OTHER.RC, and C_ERROR.RC. The sample program C_SAMPLE.RC contains most of the functions referred to in this chapter.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

Additionally, simple methods (contained on line in the module C_CIO.C) are used to manipulate character strings and text files. These methods are used only to make the programs functional. They are not provided as the recommended way of performing user input in your C programs. You should examine the needs of your application carefully and use a method that is best suited to your particular needs.

Note *Before reading this chapter, you should be familiar with the information contained in Chapter 9. The main purpose of this chapter is to provide information and examples specific to VAX C.*

17.1 The RDML/C Preprocessor Interface

When you use the RDML/C preprocessor interface, you simply include Rdb/VMS data manipulation statements directly in your program wherever you need them. When you preprocess the source program, the preprocessor converts the Rdb/VMS data manipulation statements to a series of C calls to Rdb/VMS. At run time, Rdb/VMS executes the calls and returns any retrieved data to the program.

You cannot preprocess a program that attempts to access a non-existent database, unless your database refers to the data dictionary, CDD/Plus, and refers only to the definitions stored there. That is, if you specify a compile-time file name in the DATABASE statement, the database must exist at preprocess time. If you specify a compile-time path name in the DATABASE statement, the path name element must exist in the data dictionary at preprocess time. This is because the preprocessor must be able to validate relation and field definitions in the programs that refer to the database.

Note *When you use RDML/C do not use the C string continuation character, a backslash (\ ), to continue text to a new line. RDML/C generates an error if it finds a string constant (within quotation marks) that does not begin and end on the same line. For example, the following C lines will cause a syntax error:*

```
printf ("abcdefg\
hijklmnopqrstuvwxyz");
```

Refer to Chapter 11 for information about:

- The RDML preprocessor and its command qualifiers
- C compile qualifiers like /G_FLOATING and /STANDARD=PORTABLE
- The user-defined option file that is required at the link step

Keep the following in mind when developing your programs:

- RDML does not expand or read #include files (header files). Therefore, you should not embed RDML statements in header files.
- Because you invoke the C compiler after the RDML preprocessor, you cannot define RDML statements with macros.
- The RDML/C statements are case sensitive, as is C. You must use uppercase type for the RDML statements and lowercase type for the C statements.

RDML statement syntax is described in the *RDML Reference Manual*.

17.2 Embedding RDML Statements in RDML/C Programs

RDML statements are equivalent to the Rdb/VMS data manipulation statements, which are a subset of Relational Database Operator (RDO) utility statements. With these statements you can access a database, update records, retrieve selected records, and handle Rdb/VMS exception conditions. For more information on the RDML statements and syntax, see the *RDML Reference Manual*.

17.2.1 Converting an RDO Prototype to the RDML/C Program Environment

Once you have created a prototype of your queries with the interactive RDO utility, you are ready to convert these RDO statements to the RDML/C program environment. See Chapter 7 for a discussion of creating an RDO prototype.

Example 17–1 is an RDML/C program based on the RDO prototype examples in Chapter 7.

Example 17–1 Converting an RDO Prototype to RDML/C

```
store_cand()

/* ---------------------------------------------------------- */
/* This function stores a record in the CANDIDATES relation.   */
/* It shows how to store a value in a field of VARYING STRING  */
/* data type.                                                 */
/* ---------------------------------------------------------- */
{
DECLARE_VARIABLE first_name SAME AS PERS.CANDIDATES.FIRST_NAME;
DECLARE_VARIABLE last_name SAME AS PERS.CANDIDATES.LAST_NAME;
DECLARE_VARIABLE middle_init SAME AS PERS.CANDIDATES.MIDDLE_INITIAL;

char status_info[255];

char response[80];              /* User's response from read_string() */
int succeed;                    /* Success flag                       */
int transaction_started;        /* Transaction started flag           */

first_name[0] = EOS;            /* EOS is the null terminator         */

/* Prompt user for data to store in the CANDIDATES relation */
```

(continued on next page)

Example 17–1 (Cont.) Converting an RDO Prototype to RDML/C

```
while (TRUE)
     {
      succeed = TRUE;
      response[0] = EOS;
      while (check_response (response, "Y") != 0)
          {
           printf (" \n");
           printf ("Please enter the first name of the candidate or\n");
           read_string (" type exit: ",
                        first_name, sizeof (first_name)-1);
           if (check_response (first_name, "EXIT") == 0)
                return;

           read_string ("Please enter the middle initial of the candidate: ",
                        middle_init, sizeof (middle_init)-1);

           read_string ("Please enter the last name of the candidate: ",
                        last_name, sizeof (last_name)-1);

           read_string ("Please enter the candidate status information: ",
                         status_info, 254);

           printf ("Have you entered the Candidate information \n");
           read_string ("correctly? (Y,N): ",
                        response, sizeof (response)-1);
          }
      /* Start transaction */

      transaction_started = FALSE;
      retry = 0;
      while (!transaction_started && retry <= 5)
          {
           transaction_started = TRUE;
           START_TRANSACTION READ_WRITE RESERVING CANDIDATES
                   FOR SHARED WRITE NOWAIT
               ON ERROR
                   handle_error();
                   transaction_started = FALSE;
                   retry++;
               END_ERROR;
          }

      if (!transaction_started)
          break;
/* Store the values specified by the user in the CANDIDATES       */
/* relation.  Check for errors and inform the user of the success */
/* or failure of the STORE operation.                             */

      STORE C IN CANDIDATES USING
           strcpy (C.FIRST_NAME, first_name);
           strcpy (C.LAST_NAME, last_name);
           strcpy (C.MIDDLE_INITIAL, middle_init);
           RDB$CSTRING_TO_VARYING (status_info, C.CANDIDATE_STATUS);
      END_STORE;
```

(continued on next page)

Example 17–1 (Cont.) Converting an RDO Prototype to RDML/C

```
      if (succeed == TRUE)
          {
           printf (" \n");
           printf ("Update operation succeeded\n\n");
           COMMIT;
          }
          else
          {
           printf ("Update operation failed\n\n");
           ROLLBACK;
          }

      response[0] = EOS;
     }
} /* End store_cand */
```

The syntax of RDML statements is not identical to the Rdb/VMS DML statements you may be accustomed to using in RDO and RDBPRE. When you incorporate your RDO prototype into your program, you need to remember several differences. In RDML:

- The FOR segmented string statement is used instead of the Rdb/VMS START_SEGMENTED_STRING statement to retrieve segmented strings.
- The STORE segmented string statement is used instead of the Rdb/VMS CREATE_SEGMENTED_STRING statement to store segmented strings.
- The BASED ON clause can be used to declare host language types. RDO has no equivalent statement.
- The DECLARE_VARIABLE clause can be used to declare host language variables. RDO has no equivalent clause.

See Chapter 7 for a full discussion of using prototypes in RDO and for examples of prototype queries.

17.2.1.1 Using Host Language Variables A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. You can use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following statements allow the use of host language variables:

- Any statement that permits the use of an RSE
- DATABASE (you can specify a database handle)
- GET

- READY
- FINISH

When you declare host language variables, follow the C naming rules. Ensure that the data type and size of each host language variable and its corresponding database field are compatible. Refer to Chapter 8 for the lists of equivalent C data types.

However, if you use host language variables in the form *host_variable immediately after another host language variable in an RSE, use braces around the host language statements or parentheses around the WITH clause. For example:

```
FOR D IN DEGREES WITH D.EMPLOYEE_ID = emp_id
   {
   *year_ptr = D.YEAR_GIVEN;
   }
END_FOR;
```

```
FOR D IN DEGREES WITH (D.EMPLOYEE_ID = emp_id)
   *year_ptr = D.YEAR_GIVEN;
END_FOR;
```

Host language variables within parentheses are not permitted in RDML statements (even though they are permitted in statements other than RDML statements). For example, the following syntax is not permitted in an RDML statement:

```
FOR E IN EMPLOYEES
   WITH E.LAST_NAME = (name)[offset].element
   .
   .
   .
END_FOR;
```

However, the following syntax is permitted in an RDML statement:

```
FOR E IN EMPLOYEES
  WITH E.LAST_NAME = name[offset].element
   .
   .
   .
END_FOR;
```

You can use the RDML DECLARE_VARIABLE clause to declare host language variables to ensure that the host language variable has the correct data type and size. The DECLARE_VARIABLE clause causes the RDML preprocessor to refer to a database field definition and assign the attributes of that field to the host language variable.

When you use the DECLARE_VARIABLE clause and wish to store a value in a field, be certain that text string variables are the same length as the text field in which you are storing them. Pad strings that are shorter than the text field with blank spaces; truncate strings that are longer than the text field. Strings that do not match the field in which they are stored will not be stored as expected.

Note that you should not use the DECLARE_VARIABLE clause to declare a variable to hold a segmented string field. The DECLARE_VARIABLE clause does not generate a data type for a segmented string field that is equivalent to the length of the segmented string segment; instead, the DECLARE_VARIABLE clause generates a data type that is equivalent to the logical identifier that points to a segmented string field.

The DECLARE_VARIABLE clause provides an extra character for null termination of character string variables, therefore you can terminate text string variables with the null character. For example, if the field is defined as "DATATYPE IS TEXT SIZE IS 10", then the first ten characters of the text string variable must be valid data, and the eleventh can be the null character.

Example 17–2 shows the format of the DECLARE_VARIABLE clause in RDML/C.

Example 17–2 Using DECLARE_VARIABLE to Declare a Host Language Variable in RDML/C

```
DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;
```

For more information on the DECLARE_VARIABLE clause, see Chapter 16 and the *RDML Reference Manual*.

You can use the RDML BASED ON clause to declare C typedefs and function variables, as shown in Example 17–3. The RDML BASED ON clause extracts the data type and size of a field and declares a function with the same attributes. However, in the case of TEXT or DATE fields, the BASED ON clause in RDML/C returns a pointer to a character.

Example 17–3 Using the BASED ON Clause in RDML/C

```
typedef BASED ON JOBS.JOB_CODE  job_code_type;
typedef BASED ON JOBS.JOB_TITLE job_title_type;
```

Also, you can declare host language variables by copying database definitions from the data dictionary, CDD/Plus, using the C #dictionary statement.

When you use the #dictionary control line in C to declare a variable for a TEXT field, be aware that you cannot use a strcpy or similar function to copy strings into the variable. The variable declared by the #dictionary control line does not leave space for the null terminator that is conventionally used to terminate strings in C programs. However, if you use the RDML DECLARE_VARIABLE clause instead of the #dictionary control line, you do not need to be concerned with this issue. The DECLARE_VARIABLE clause declares the variable for a TEXT field with an extra space provided for the null terminator.

See Chapter 16 for more information on using the data dictionary to declare host language variables in RDML/C programs.

You can use simple and complex C host language variables, such as arrays or typedefs, in an RSE. However, do not use functions or procedures within the RSE. For example, the following RDML/C code does not preprocess:

```
/* bad code, won't preprocess correctly! */
    FOR FIRST 5 E IN EMPLOYEES WITH E.LAST_NAME = strcat("Black", "-Smith");
    printf ("%s/n",E.LAST_NAME);
    END_FOR;
```

However, you can assign the result of a function to a variable and use the variable within the RSE.

17.2.1.2 Converting DATE Data Type to TEXT DATE data types are stored in Rdb/VMS databases in encoded binary format. To display a date, your program must first retrieve the binary value and convert it to an ASCII string. This is done by using the VMS system service routine, SYS$ASCTIM, to perform the conversion.

See the *VMS System Services Volume* for more information on using SYS$ASCTIM.

Example 17–4, a code fragment from the ADD_EMPLOYEES function, demonstrates how to display a date:

Example 17–4 Using SYS$ASCTIM System Service Routine in RDML/C

```
FOR E IN EMPLOYEES
  WITH E.RDB$DB_KEY = rdb_key_array[x]
  ON ERROR
    handle_error();
  END_ERROR
  printf ("%s %s. ", E.FIRST_NAME, E.MIDDLE_INITIAL);
  printf ("%s\n", E.LAST_NAME);
  printf ("%s", E.ADDRESS_DATA_1);
  printf ("%s\n", E.ADDRESS_DATA_2);
  printf ("%s %s\n", E.CITY, E.STATE);
  printf ("%s\n", E.POSTAL_CODE);

  /* Convert binary date to ascii date */

  stat = SYS$ASCTIM( &ascii_birthday.dsc$w_length,
                     &ascii_birthday,
                     employee_record.birthday,
                    0);
  if ( (stat & 1) != 1 )
     printf ("Data conversion failed");
  else
    {
    ascii_birthday.dsc$a_pointer[ascii_birthday.
    dsc$w_length] =’\0’;
    puts (ascii_birthday.dsc$a_pointer);
    }
END_FOR;
```

17.2.1.3 Converting ASCII DATE Strings to Binary Format Use the VMS system service routine, SYS$BINTIM, to convert ASCII DATE strings into a binary representation so the DATE data type fields can be stored in the database.

See the *VMS System Services Volume* for more information on using SYS$BINTIM.

Example 17–5, a code fragment from the ADD_EMPLOYEES function, demonstrates how to use SYS$BINTIM in an RDML/C program.

Example 17–5 Using the SYS$BINTIM System Service Routine in RDML/C

```
printf ("Please enter the Employee’s birthday\n");
read_string ("In the format: dd-MMM-yyyy :",
              ascii_birthday.dsc$a_pointer,
              strlen(ascii_birthday.dsc$a_pointer) );

/* Convert ASCII date to binary date. */

stat   = SYS$BINTIM(&ascii_birthday, employee_record.birthday);
  if ( (stat & 1) == 1 )
     break;
  else
     {
     ascii_birthday.dsc$a_pointer[ascii_birthday.dsc$w_length] = ’\0’;
    printf ("***Invalid date ’%s’ ****\n",
             ascii_birthday.dsc$a_pointer);
     }
```

17.2.2 Using Literals

Use literal values to replace variables in the same way you would in any high-level language. Literal values can be either numeric or character strings. A string literal must be quoted in double quotation marks (" ") in C and, if you use it to replace a field value, or make a comparison to a field value, it must be padded with blanks to fit the defined size of the field to which it refers.

Notice in the following example that the literal “Typist” is padded with blanks to fit the defined size of the JOB_TITLE field. You must pad strings this way, or by using a function, to ensure that string values are stored in the database correctly.

```
STORE J IN JOBS USING
   strcpy (J.JOB_CODE, "TYPS");
   strcpy (J.JOB_TITLE, "Typist              ");
   J.MAXIMUM_SALARY =  15000;
   J.MINIMUM_SALARY =  20000;
   strcpy (J.WAGE_CLASS, "3");
END_STORE;
```

17.2.3 Forming Record Streams

In C, and any language that you use to access an Rdb/VMS database, you select the records you are interested in manipulating by gathering records into a stream. You create this stream using the RDML statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

17.2.4 Retrieving Records

RDML provides you with three statements to retrieve records:

- FOR
- Two START_STREAM statements:
  - Declared START_STREAM
  - Undeclared START_STREAM

17.2.4.1 Using the FOR Statement to Retrieve Records The FOR statement forms a record stream and provides automatic iteration for any RDML and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

Example 17–6 shows a FOR statement from the DISPLAY_CAND function. It uses the flag "succeed" to determine if the RSE has been satisfied. If a candidate record is found with field values that match the values in the host language variables, the succeed flag is set to true. If no record matches the values in the host language variables, then the succeed flag remains set to false. In C, you must use the RDB$VARYING_TO_CSTRING macro to retrieve a field of VARYING STRING data type. See Section 17.2.6.4 for information on using the VARYING STRING data type in C programs.

Example 17–6 Using the FOR Statement in RDML/C

```
succeed = FALSE;
FOR C IN CANDIDATES WITH C.FIRST_NAME = first_name
   AND C.MIDDLE_INITIAL = middle_init
   AND C.LAST_NAME = last_name
     RDB$VARYING_TO_CSTRING (C.CANDIDATE_STATUS, status_info);
     printf ("%s %s\n",
             C.FIRST_NAME, C.LAST_NAME);
     printf ("has the following status: %s\n\n", status_info);
             succeed = TRUE;
END_FOR;
```

17.2.4.2 Using Declared Streams to Retrieve Records RDML provides two forms of the START_STREAM statement, the *declared* and the *undeclared* START_STREAM statements. Declared streams provide all the features of the undeclared streams and more. Most importantly, undeclared streams require that the statements you use to manipulate the stream be enclosed by the START_STREAM and END_STREAM statements in your source program. Declared streams do not impose this restriction. The statements you use to manipulate the stream may appear in any order within your program as long as the DECLARE_STREAM statement appears first and the statements execute in a logical order (START_STREAM, FETCH, GET, END_STREAM).

Digital recommends that all new applications use the declared START_STREAM statement. For this reason, only the declared START_STREAM statement is discussed in this section. Complete details on the differences between declared and undeclared START_STREAM statements are provided in Chapter 9.

Example 17–7, from the PAIR function, shows the use of the declared START_STREAM statement and the FETCH statement. The example pairs a CANDIDATES record with an EMPLOYEES record at random.

Example 17–7 Using the Declared START_STREAM and FETCH Statements in RDML/C

```
/* Declarations for the function named PAIR                          */
/* Declare two streams: one for the CANDIDATES relation and          */
/* the other for the EMPLOYEES relation.                             */

DECLARE_STREAM emps USING EM IN EMPLOYEES SORTED BY EM.FIRST_NAME;
DECLARE_STREAM cands USING CA IN CANDIDATES SORTED BY CA.LAST_NAME;

/* Flags for end-of-stream condition */

globaldef    int  end_of_emps  = FALSE;
globaldef    int  end_of_cands = FALSE;
   .
   .
   .
pair()

/* ----------------------------------------------------------------*/
/* This function demonstrates the use of the declared START_STREAM  */
/* statement.  The output of this program is merely a random        */
/* matching of each CANDIDATES record with an EMPLOYEES record.     */
/* The functions called in this function appear just after this one.*/
/* ----------------------------------------------------------------*/

{
char response[80];           /* User's Response from read_string() */

START_TRANSACTION READ_ONLY;

/* Open both streams and set flags for the end-of-stream condition  */
/* to false.                                                         */

    open_candidates();
    open_employees();
    end_of_emps =  FALSE;
    end_of_cands = FALSE;

/* Fetch a record from the CANDIDATES and EMPLOYEES relations.      */

    read_a_candidate();
    read_an_employee();

/* Print the employee and candidate names until the end-of-stream   */
/* condition is met for the stream of CANDIDATES records.           */
```

(continued on next page)

Example 17–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDML/C

```
    while (!end_of_cands)
        {
         printf ("%s %s     %s %s\n", EM.LAST_NAME, EM.FIRST_NAME,
                                      CA.LAST_NAME, CA.FIRST_NAME );
         read_a_candidate();
         if (!end_of_cands)
           {
           read_an_employee();
           }
        }

    printf (" \n");
    read_string("Press RETURN to continue",
                 response, sizeof (response)-1);

    /* Close both streams. */

    close_employees();
    close_candidates();
    COMMIT;
} /*  End pair */

/* These functions control streams in the PAIR function.            */
/* Of course, a simple program such as this does not require the    */
/* use of functions to separate the RDML statements.  It is done    */
/* here to demonstrate what you can do. Note that the statements do */
/* not appear in the order that they will be executed.  This is a   */
/* feature that declared streams have and undeclared streams do     */
/* not have.                                                        */

read_a_candidate( )
{
   FETCH cands
      AT END
        end_of_cands = TRUE;
   END_FETCH;
}

open_candidates( )
{
START_STREAM cands;
}

open_employees( )
{
START_STREAM emps;
}

read_an_employee( )
{
FETCH emps
   AT END
     end_of_emps = TRUE;
   END_FETCH;
}
```

(continued on next page)

Example 17–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDML/C

```
close_employees( )
{
END_STREAM emps;
}

close_candidates( )
{
END_STREAM cands;
}
```

17.2.5 Retrieving Segmented Strings

Retrieving segmented strings is a two-step process. First, you must retrieve the record that contains the segmented string field; then, you must retrieve the individual segments that make up the segmented string field.

You may find it easier to picture a segmented string by referring to Figure 8–1 in Chapter 8.

RDML provides you with the FOR statement with segmented strings to retrieve segmented strings. You must use two streams when processing segmented string streams. Use the first FOR (or START_STREAM) statement to form an outer stream of records, and then use a second FOR statement to form an inner stream of segments. This inner stream identifies the segments contained in the field specified by the first RSE. Use different context variables for the inner and outer streams.

Remember that to retrieve a segmented string, you must begin at the first segment and retrieve segments in the order in which they are stored, that is, sequentially.

Example 17–8 from the DISPLAY_RESUME function:

- Uses a FOR statement to search the database for a record with a value for the EMPLOYEE_ID field that matches the host language variable, employee_id
- Uses a second FOR statement to loop through the segments of the segmented string field for the selected EMPLOYEES record
- Uses a printf statement to retrieve field values, the individual segments that make up the segmented string
- Displays these values on the terminal

Example 17–8 Using the FOR Statement with Segmented Strings in RDML/C

```
display_resume()

/* -------------------------------------------------------------*/
/* This function demonstrates how to retrieve a field of data   */
/* type SEGMENTED STRING.                                       */
/* -------------------------------------------------------------*/

{
DECLARE_VARIABLE employee_id SAME AS RESUMES.EMPLOYEE_ID;

char response[80];        /* User’s response from read_string() */
int succeed;
employee_id[0]= EOS;      /* EOS is the null terminator         */


/* Prompt the user to enter the ID of the employee resume that  */
/* he or she wants to view.  If user enters ’exit’ then exit    */
/* function.                                                    */

while (TRUE)
   {
   response[0] = EOS;
   while (check_response (response, "Y") != 0)
      {
      printf (" \n");
      printf ("Please enter the ID number of the Employee whose\n");
      read_string (" resume you want to display or type exit: ",
                     employee_id, sizeof (employee_id)-1);
      if (check_response (employee_id, "EXIT") == 0)
        return;

      read_string ("Have you entered all the data correctly? (Y,N) ",
                   response, sizeof (response)-1);
      }
START_TRANSACTION READ_ONLY RESERVING RESUMES FOR SHARED READ;

/* Start an outer FOR loop to retrieve the employee record(s)   */
/* with the specified ID.                                       */

     succeed = FALSE;
     FOR R2 IN RESUMES WITH R2.EMPLOYEE_ID = employee_id
        succeed = TRUE;

        /* Start an inner FOR loop to retrieve the segments of  */
        /* the segmented string that make up the employee’s     */
        /* resume.  Display each segment as it is retrieved     */
        /* from the database.                                   */


        FOR TEXT IN R2.RESUME
           printf ("%.*s\n", TEXT.LENGTH,TEXT.VALUE);
        END_FOR;
     END_FOR;
```

(continued on next page)

Example 17–8 (Cont.) Using the FOR Statement with Segmented Strings in RDML/C

```
/* If a record with the specified ID was not found */
/* then inform the user.                           */

     if (succeed == FALSE)
        {
        printf("Employee: %s%s",employee_id,"has no resume on file");
        }
   COMMIT;
   employee_id[0] = EOS;
   }
} /* End display_resume */
```

17.2.6 Retrieving Field Values

RDML lets you use several methods to retrieve field values, as outlined in the following list:

- Use the GET statement to retrieve any value including statistical values and the results of conditional expressions from the database.
- Use the C assignment statement or a C function to retrieve one, several, or all the fields in a database record and assign those values to one or more host language variables.
- Refer to a field as a parameter of a function.
- Use the printf or other input/output function to print out database values.

Although you can use an assignment statement to retrieve statistical values and the results of conditional expressions from the database, Digital recommends that you always use the GET statement in these cases. The GET statement lets you perform error checking with the ON ERROR clause, a clause that is not available in statistical functions and conditional expressions. Furthermore, a function call is generated by an assignment statement that is not generated when you use the GET statement. Therefore, the GET statement is more efficient than an assignment statement in the context of statistical and conditional expressions.

Section 17.2.6.1, Section 17.2.6.2, and Section 17.2.6.4 discuss retrieving field values. Section 17.2.6.3 discusses retrieving statistical values.

17.2.6.1 Using an Assignment Statement to Retrieve Field Values

When you form a record stream using the FOR statement, you can assign database values to host language variables within the FOR . . . END_FOR block. You can also write these values using the printf statement.

Example 17–9, from the LIST_RECORD function, demonstrates how to use the C printf statement to retrieve database values in C.

Example 17–9 Using an Assignment Statement to Retrieve Field Values in RDML/C

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
   FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      printf ("Name is: %s %s\n", E.FIRST_NAME, E.LAST_NAME);
      printf ("Degree is: %s\n", D.DEGREE);
      printf ("Degree field is: %s\n\n", D.DEGREE_FIELD);
   END_FOR;
    .
    .
    .
END_FOR;
```

When you form a record stream using the START_STREAM statement, you include the FETCH and GET or assignment statements within the START_STREAM . . . END_STREAM block.

See Example 17–7 for an example of using the FETCH and assignment statements within a START_STREAM . . . END_STREAM block.

17.2.6.2 Using the GET * Statement to Retrieve Field Values

A special form of the GET statement is the GET * statement, which lets you retrieve database values at the record level rather than the field level. You can retrieve all the fields in a record from a relation with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the database relation, with record field names that match the relation field names. The GET * statement in the following example (from the ADD_EMPLOYEES function) retrieves all of the fields in an EMPLOYEES record and places their values in the employee_record host language record structure.

```
static struct
    {
DECLARE_VARIABLE OF employee_id SAME AS PERS.EMPLOYEES.EMPLOYEE_ID;
DECLARE_VARIABLE OF last_name SAME AS PERS.EMPLOYEES.LAST_NAME;
DECLARE_VARIABLE OF first_name SAME AS PERS.EMPLOYEES.FIRST_NAME;
DECLARE_VARIABLE OF middle_initial SAME AS PERS.EMPLOYEES.MIDDLE_INITIAL;
DECLARE_VARIABLE OF address_data_1 SAME AS PERS.EMPLOYEES.ADDRESS_DATA_1;
DECLARE_VARIABLE OF address_data_2 SAME AS PERS.EMPLOYEES.ADDRESS_DATA_2;
DECLARE_VARIABLE OF city SAME AS PERS.EMPLOYEES.CITY;
DECLARE_VARIABLE OF state SAME AS PERS.EMPLOYEES.STATE;
DECLARE_VARIABLE OF postal_code SAME AS PERS.EMPLOYEES.POSTAL_CODE;
DECLARE_VARIABLE OF sex SAME AS PERS.EMPLOYEES.SEX;
DECLARE_VARIABLE OF status_code SAME AS PERS.EMPLOYEES.STATUS_CODE;
DECLARE_VARIABLE OF birthday SAME AS PERS.EMPLOYEES.BIRTHDAY;
    } employee_record;
              .
              .
              .
     FOR FIRST 1 E IN EMPLOYEES
         GET
             employee_record = E.*;
         END_GET;
     END_FOR;
```

17.2.6.3 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly without processing each record in the record stream. The result of a statistical expression is an aggregate, and the data type of the result is often not the same data type as the field on which the statistical expression is performed. See Chapter 8 for information on the data type conversions performed by statistical expressions.

There are two advantages to using a GET rather than an assignment statement. First, the GET statement supports the ON ERROR . . . END_ERROR clause, which allows you to detect errors that occur during the statistical or Boolean function. Second, using the GET statement results in more efficient code than an assignment statement when it is used with statistical and Boolean functions.

Example 17–10, from the STATS function, uses the COUNT statistical function to find the total number of records in the EMPLOYEES relation.

Example 17–10 Using the GET Statement to Retrieve Statistical Values in RDML/C

```
stats()

/* ----------------------------------------------------------- */
/* This function displays the total number of records stored   */
/* in the EMPLOYEES relation.                                  */
/* ----------------------------------------------------------- */
{
char response[80];       /* User's response from read_string() */
int atotal;              /* Total                              */

START_TRANSACTION READ_ONLY;

/* Use the GET statement with a statistical expression to      */
/* calculate the total number of records in the EMPLOYEES      */
/* relation.                                                   */

     printf ("\n\n");
     printf ("The number of employees in the Corporation is: ");

     GET
      atotal =  (COUNT OF E IN EMPLOYEES);
     END_GET;

     printf ("%d\n",atotal);
     read_string ("Press RETURN to continue",
                  response, sizeof (response)-1);
COMMIT;

} /* End stats */
```

17.2.6.4 Retrieving Field Values of the VARYING STRING Data Type To retrieve the value of a VARYING STRING database field, you must use the C macro RDB$VARYING_TO_CSTRING (supplied by Rdb/VMS) within the FOR or START_STREAM statement.

Example 17–11 shows a program that uses the RDB$VARYING_TO_CSTRING macro to retrieve the VARYING STRING value stored in the CANDIDATE_STATUS field of the CANDIDATES relation.

Example 17–11 Retrieving Field Values of the VARYING STRING Data Type in RDML/C

```
#include <stdio.h>
DATABASE PERS = FILENAME "MF_PERSONNEL";

main()
{
char candidate_status[255];

READY PERS;
START_TRANSACTION READ_ONLY;

FOR C IN CANDIDATES
  printf("%s  %s  %s\n", C.FIRST_NAME, C.MIDDLE_INITIAL, C.LAST_NAME);
  RDB$VARYING_TO_CSTRING(C.CANDIDATE_STATUS,candidate_status);
  printf("%s\n\n", candidate_status);
END_FOR;

COMMIT;
FINISH;
}
```

17.2.7 Updating Records Using the STORE, MODIFY, and ERASE Statements

The RDML update statements can only be used within a read/write transaction. (You may, of course, include any valid RDML statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

If you update a record and triggered actions have been defined for the relation containing the record, the update operation (STORE, MODIFY, or ERASE) will have the specified effect on all the relations in the database that have a foreign key relationship with the record you want to update.

If a relation-specific constraint has been defined, your ability to perform update operations may depend on the presence of matching field values in other relations. For more information on relation-specific constraints, see Section 6.6.

Note *You may not use a view to update records if that view refers to more than one relation.*

17.2.7.1 Storing Records

You can insert values into one or more fields in one record using a single STORE statement. To store more than one record in a relation, include the STORE statement within a program loop.

Note that RDML may return unpredictable results when a C multipath statement, such as the C switch statement, is embedded in an RDML STORE statement. The problem occurs when a field is referred to but not used at run time. This is because RDML assumes that any field mentioned within a STORE . . . END_STORE block is going to be updated.

In the following example, if the program falls through to case 2 at run time, a value will be stored in the FIRST_NAME field even though FIRST_NAME is not referred to in case 2. Upon seeing the field referred to in case 1, RDML sets up a buffer for both the FIRST_NAME and LAST_NAME fields. Because case 2 does not supply data for the FIRST_NAME field, RDML sends to the database whatever happens to be in the buffer for the FIRST_NAME field.

The following code will cause unpredictable results:

```
STORE E IN EMPLOYEES USING
  switch (i){
   case ’1’:
      strcpy (E.LAST_NAME,"Smith         ");
      strcpy (E.FIRST_NAME,"Andrew    ");
      break;
   case ’2’:
      strcpy (E.LAST_NAME, "Jones         ");
     break;
   }
END_STORE;
```

When different fields are referred to in a multipath statement, the RDML statement should be embedded in the host language multipath statement as shown in the following example:

```
     switch(i) {
       case ’1’:
            STORE E IN EMPLOYEES USING
               strcpy (E.LAST_NAME,"Smith         ");
               strcpy (E.FIRST_NAME,"Andrew    ");
            END_STORE;
       break;
       case ’2’:
             STORE E IN EMPLOYEES USING
                strcpy (E.LAST_NAME, "Jones         ");
            END_STORE;
       break;
       }
END_FOR;
```

Example 17–12, from the STORE_CAND function, stores a candidate's record in the CANDIDATES relation.

Example 17–12 Storing Records in RDML/C

```
transaction_started = FALSE;
retry = 0;
while (!transaction_started && retry <= 5)
   {
   transaction_started = TRUE;
   START_TRANSACTION READ_WRITE RESERVING CANDIDATES
        FOR SHARED WRITE NOWAIT
      ON ERROR
        handle_error();
        transaction_started = FALSE;
        retry++;
      END_ERROR;
    }

    if (!transaction_started)
       break;

/* Store the values specified by the user in the CANDIDATES       */
/* relation.  Check for errors and inform the user of the success */
/* or failure of the STORE operation.                             */

    STORE C IN CANDIDATES USING
       strcpy (C.FIRST_NAME, first_name);
       strcpy (C.LAST_NAME, last_name);
       strcpy (C.MIDDLE_INITIAL, middle_init);
       RDB$CSTRING_TO_VARYING (status_info, C.CANDIDATE_STATUS);
    END_STORE;

    if (succeed == TRUE)
       {
       printf (" \n");
       printf ("Update operation succeeded\n\n");
       COMMIT;
       }
    else
       {
       printf ("Update operation failed\n\n");
       ROLLBACK;
       }
```

17.2.7.1.1 Using the STORE * Statement to Store Records A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in a record with the STORE * statement. To use the STORE * statement, you must first declare a record structure that specifies all the fields in the relation definition, with C record field names that match the database field names exactly. Then, put the values you want to store in the database record fields into the C program record and store the entire C record using the

STORE * statement. Example 17–13 shows the use of the STORE * statement to store the fields in the employee_record record structure that in turn is stored in the EMPLOYEES relation of the MF_PERSONNEL database.

Example 17–13 Using the STORE * Statement in RDML/C

```
/* Declare a C record structure. */

static struct
    {
DECLARE_VARIABLE OF employee_id SAME AS PERS.EMPLOYEES.EMPLOYEE_ID;
DECLARE_VARIABLE OF last_name SAME AS PERS.EMPLOYEES.LAST_NAME;
DECLARE_VARIABLE OF first_name SAME AS PERS.EMPLOYEES.FIRST_NAME;
DECLARE_VARIABLE OF middle_initial SAME AS PERS.EMPLOYEES.MIDDLE_INITIAL;
DECLARE_VARIABLE OF address_data_1 SAME AS PERS.EMPLOYEES.ADDRESS_DATA_1;
DECLARE_VARIABLE OF address_data_2 SAME AS PERS.EMPLOYEES.ADDRESS_DATA_2;
DECLARE_VARIABLE OF city SAME AS PERS.EMPLOYEES.CITY;
DECLARE_VARIABLE OF state SAME AS PERS.EMPLOYEES.STATE;
DECLARE_VARIABLE OF postal_code SAME AS PERS.EMPLOYEES.POSTAL_CODE;
DECLARE_VARIABLE OF sex SAME AS PERS.EMPLOYEES.SEX;
DECLARE_VARIABLE OF status_code SAME AS PERS.EMPLOYEES.STATUS_CODE;
DECLARE_VARIABLE OF birthday SAME AS PERS.EMPLOYEES.BIRTHDAY;
    } employee_record;
 . . .
/* Assign values to the host language variables. */

read_string ("Please enter the Employee’s last name: ",
              employee_record.last_name,
              sizeof (employee_record.last_name)-1);

read_string ("Please enter the Employee’s first name: ",
              employee_record.first_name,
              sizeof (employee_record.first_name)-1);
 . . .
/* Store these values using the STORE * syntax. */

STORE E IN EMPLOYEES USING
   ON ERROR
      succeed = FALSE;
      handle_error();
   END_ERROR;
   E.* = employee_record;
 . . .
END_STORE;
```

17.2.7.1.2 Storing VARYING STRING Data Types in the Database

You must use the C macro RDB$CSTRING_TO_VARYING (supplied by Rdb/VMS) to store values for VARYING STRING fields in the database. Example 17–14 shows a fragment from the STORE_CAND function that uses the RDB$CSTRING_TO_VARYING macro to store VARYING STRING data in the CANDIDATES relation.

Example 17–14 Storing VARYING STRING Data in RDML/C

```
STORE C IN CANDIDATES USING
   strcpy (C.FIRST_NAME, first_name);
   strcpy (C.LAST_NAME, last_name);
   strcpy (C.MIDDLE_INITIAL, middle_init);
   RDB$CSTRING_TO_VARYING (status_info, C.CANDIDATE_STATUS);
END_STORE;
```

17.2.7.1.3 Using the STORE Statement with Segmented Strings to Store Segmented Strings The STORE segmented string statement behaves in a similar manner to the FOR segmented string statement. You must use two streams when you process segmented string streams. Use the first STORE statement to form an outer stream of records, and then use the second STORE statement to form an inner stream of segments. This second STORE statement identifies the segments that are contained in the field specified by the first STORE statement. Use a different context variable in each of the two STORE statements.

Note that the inner STORE statement uses a segmented string variable in place of the context variable, and that the field name is qualified by the context variable specified in the outer STORE statement. Your program must explicitly repeat the inner STORE statement to store individual segments, or provide iteration for an inner STORE loop.

Note *See Section 9.2.6.1.2 for information about defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name with an appropriate value for storing your segmented strings.*

Note *Segmented strings cannot be updated (ERASE, MODIFY, or STORE) as part of a triggered action. For more information, see the DEFINE TRIGGER statement in the* VAX Rdb/VMS RDO and RMU Reference Manual.

Example 17–15, from the STORE_RES function, demonstrates how to store a segmented string in C.

Example 17–15 Storing a Segmented String in RDML/C

```
store_res()

/************************************************************/
/* This function demonstrates how to store a record with    */
/* a field of data type SEGMENTED STRING.                   */
/************************************************************/

{
DECLARE_VARIABLE employee_id SAME AS RESUMES.EMPLOYEE_ID;

char response[80];           /* User's response from read_string() */
char my_file[21];            /* Resume file                        */
char buffer[80];             /* Temporary buffer for fgets         */
FILE *fopen(), *fp;          /* File pointer                       */

employee_id[0] = EOS;        /* EOS is the null terminator         */
while (TRUE)
      {
      response[0] = EOS;
      while (check_response (response, "Y") != 0)
         {
          printf (" \n");

          /* Prompt user for employee ID of the EMPLOYEES   */
          /* record that he or she wants to store.          */

          printf ("Please enter the ID number of the Employee\n");
          read_string (" or type exit: ",
                       employee_id, sizeof (employee_id)-1);
          if (check_response (employee_id, "EXIT") == 0)
               return;
          /* Prompt user for the file name of the resume    */
          /* to be stored.                                  */

          read_string ("Please enter file name of the resume: ",
                       my_file, sizeof (my_file)-1);
          read_string ("Have you entered all the data correctly? (Y,N) ",
                       response, sizeof (response)-1);
         }
      fp = fopen(my_file, "r");

      START_TRANSACTION READ_WRITE RESERVING RESUMES
                        FOR SHARED WRITE;

/* Use the STORE statement with segmented strings to store the */
/* record.  The outer STORE statement creates the new RESUMES  */
/* record.  The inner STORE stores the individual segments of  */
/* the SEGMENTED STRING field.                                 */
```

(continued on next page)

Example 17–15 (Cont.) Storing a Segmented String in RDML/C

```
               STORE R IN RESUMES USING
               ON ERROR
                    handle_error();
               END_ERROR

               strcpy (R.EMPLOYEE_ID, employee_id);
               while (fgets (buffer, 132, fp) != NULL)
                    {
                     STORE LINE IN R.RESUME
                          strcpy (LINE.VALUE, buffer);
                          LINE.LENGTH = strlen (buffer)-1;
                     END_STORE;
                    }
           END_STORE;
           fclose (fp);

      COMMIT;
      response[0] = EOS;
     }

} /* End store_res */
```

17.2.7.2 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of a record in a relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Example 17–16, from the MODIFY_ADDRESS function, modifies a record in the EMPLOYEES relation. The values used to modify the record were requested earlier in the program.

Example 17–16 Modifying Records in RDML/C

```
        .
        .
        .
START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR SHARED WRITE;

/* Modify the address fields for the specified EMPLOYEES record. */

   FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
      MODIFY E USING
         ON ERROR
            succeed = FALSE;
            handle_error();
         END_ERROR

         strcpy (E.ADDRESS_DATA_1, street);
         strcpy (E.ADDRESS_DATA_2, address_data);
         strcpy (E.CITY, town);
         strcpy (E.STATE, state);
         strcpy (E.POSTAL_CODE, postal_code);
      END_MODIFY;
   END_FOR;
```

17.2.7.2.1 Using the MODIFY * Statement to Modify Records A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement.

Only use the MODIFY * statement if you need to modify every field value in a record. Modifying a field by replacing one value with an identical value, needlessly adds overhead to your program. For example, your program may check constraints on a field value that *you know* is valid because it is the same value that the field presently holds.

Example 17–17 replaces the field values of an employee record in the JOB_ HISTORY relation with the values in the job_history host language record structure.

Example 17–17 Using the MODIFY * Statement in RDML/C

```
FOR J IN JOB_HISTORY WITH
  J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
  AND J.JOB_END MISSING
     MODIFY J USING
         J.* = job_history;
     END_MODIFY
END_FOR
```

17.2.7.2.2 Modifying Segmented Strings The method you use to modify a segmented string involves two RDML statements: the MODIFY statement and the STORE statement with segmented strings. The MODIFY statement selects the records for which you want to modify the segmented string field. An inner STORE statement with segmented strings deletes the existing segmented string and writes over the existing segmented string handle with a new segmented string handle. Note that you cannot modify the individual segments that make up the segmented string; you must replace the entire segmented string.

Example 17–18 demonstrates how to modify a segmented string in RDML/C.

Example 17–18 Modifying Segmented String Fields in RDML/C

```
mod_resume()

/* -------------------------------------------------------------*/
/* This function demonstrates how to modify a field of data    */
/* type SEGMENTED STRING.                                       */
/* -------------------------------------------------------------*/

{
DECLARE_VARIABLE employee_id SAME AS RESUMES.EMPLOYEE_ID;

char response[80];       /* User's response from read_string() */
char my_file[21];        /* Resume file                        */
char buffer[80];         /* Temporary buffer for fgets         */
FILE *fopen(), *fp;      /* File pointer                       */

employee_id[0] = EOS;    /* EOS is the null terminator         */

/* Prompt user for the employee ID of the RESUMES record he    */
/* or she wants to modify.                                     */

while (TRUE)
  {
  response[0] = EOS;
  while (check_response (response, "Y") != 0)
     {
     printf (" \n");
     printf ("Please enter the ID number of the Employee\n");
     read_string (" or type exit: ",
                    employee_id, sizeof (employee_id)-1);
       if (check_response (employee_id, "EXIT") == 0)
            return;
```

(continued on next page)

Example 17–18 (Cont.) Modifying Segmented String Fields in RDML/C

```
/* Prompt user for the file name of the resume that will replace  */
/* the old resume.                                                */

       printf ("To modify a resume, you must supply a new file");
       printf ("name that contains the new resume.\n");
       printf (" resume and replace it with a new resume\n");
       read_string ("Please enter file name of new resume: ",
                    my_file, sizeof (my_file)-1);
       read_string ("Have you entered all the data correctly? (Y,N) ",
                     response, sizeof (response)-1);
     }

     fp = fopen(my_file, "r");

     START_TRANSACTION READ_WRITE RESERVING RESUMES
                       FOR SHARED WRITE;

/* Start an outer FOR loop to retrieve the employee record(s)     */
/* with the specified ID.                                         */

        FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id

/* Use a MODIFY statement to change the value of the segmented    */
/* string field.                                                  */

           MODIFY R USING
              ON ERROR
                 handle_error();
              END_ERROR;

/* Read in the new resume and use a STORE operation to store      */
/* a new segmented string handle in the RESUMES relation.         */

              while (fgets (buffer, 132, fp) != NULL)
                    {
                    /* fgets reads a carriage return if it exists */
                    /* in the file into the buffer - therefore    */
                    /* subtract 1.                                */

                     STORE SEG IN R.RESUME
                          strcpy(SEG.VALUE, buffer);
                          SEG.LENGTH = strlen(buffer) - 1;
                     END_STORE;
                    }
            END_MODIFY;
         END_FOR;
         fclose (fp);

      COMMIT;
      response[0] = EOS;
     }

} /* End mod_resume */
```

17.2.7.3 Erasing Records You can delete one, many, or all the records from a relation using the ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream containing the records you wish to erase.

Example 17–19, from the DELETE_RECORD function, demonstrates how to erase records in RDML/C programs.

Note *The definition of the sample personnel database includes the trigger EMPLOYEE_ID_CASCADE_DELETE, which performs an automatic deletion of records in the relations named in ERASE statements in Example 17–19 (except for RESUMES) when the record with the matching employee ID is deleted from the EMPLOYEES relation. Thus, you would not need to include "cascading deletion" logic in your programs if it were already included in a trigger definition.*

Example 17–19 Erasing Records in RDML/C

```
/* Earlier in the function DELETE_RECORD, an employee record was   */
/* retrieved to make certain that the user wants to delete this    */
/* employee's records. Having made that determination, the program */
/* will now delete all records associated with that employee. When */
/* the employee record was retrieved, the database key associated  */
/* with that record was also retrieved.  It can be used here to    */
/* quickly locate that employee's EMPLOYEES record again, so that  */
/* records for this employee can be erased from all the relations  */
/* in which he or she has a record.                                */

START_TRANSACTION READ_WRITE RESERVING EMPLOYEES,
   SALARY_HISTORY, JOB_HISTORY, DEGREES, RESUMES FOR
   SHARED WRITE;

   FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = db_key
      ERASE E;
   END_FOR;

   FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = employee_id
      ERASE JH;
   END_FOR;

   FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = employee_id
      ERASE SH;
   END_FOR;

   FOR D IN DEGREES WITH D.EMPLOYEE_ID = employee_id
      ERASE D;
   END_FOR;

   FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id
      ERASE R;
   END_FOR;
```

17.3 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer or address that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this key as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, the scope of a dbkey ends with a COMMIT statement. That is, a dbkey is guaranteed to point to the same record for the life of the transaction in which it is retrieved.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

The following example demonstrates how to specify the dbkey scope in an RDML/C program.

```
DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH;
```

Suggestions on how you can take advantage of the dbkey scope are contained in Section 9.2.7.

17.4 Using Structured Programming

Programs and modules that pass through the RDML preprocessor do not have unlimited freedom in structure. Calls to routines, subprograms, and functions require that you pay special attention to the context from which they are called.

Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- FOR
- GET
- DECLARE_STREAM
- START_STREAM
- END_STREAM
- FETCH
- STORE

- MODIFY
- ERASE
- STORE statement with segmented strings
- FOR statement with segmented strings

These individual data manipulation statements each form only part of a complex call to the database. The preprocessor generates one call to the database, using more than one data manipulation statement. For example, MODIFY statements execute within the context of a FOR statement or an undeclared START_STREAM statement. The database update can be made only by using both the FOR or undeclared START_STREAM statement and the MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. However, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

Also keep in mind that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement, and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which it is issued.

A stream declared with the DECLARE_STREAM statement lets you place the stream manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

For more information on the declared and undeclared START_STREAM statement, see Section 9.2.3.2. Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- DATABASE
- READY
- START_TRANSACTION
- GET

- COMMIT
- ROLLBACK
- FINISH
- DECLARE_STREAM

Remember that you must issue the DECLARE_STREAM statement before you can issue a declared START_STREAM statement, and the DATABASE statement must appear in the data declaration section of your program.

Example 17–20, from the DELETE_RECORD and CALL_OTHER functions, demonstrates structured programming in an RDML/C program. The DELETE_RECORD and CALL_OTHER functions are in modules that are separately preprocessed and processed. They are linked with the LINK command. The DELETE_RECORD function passes the value of a dbkey to the CALL_OTHER function. This function finds the record associated with the dbkey and displays this record on the terminal. Although it is not necessary to program this query in two modules, it is done here to demonstrate how to pass variables between separately processed modules.

Example 17–20 Using Structured Programming in RDML/C

```
Function DELETE_RECORD:

START_TRANSACTION (TRANSACTION_HANDLE trans_1 ) READ_WRITE;

/* Find the employee record that the user wants to delete.  If      */
/* an error occurs during the FOR operation, call an error handler. */

   FOR (TRANSACTION_HANDLE trans_1)
      E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
     ON ERROR
        handle_error();
     END_ERROR

     /* Get the dbkey of the EMPLOYEES record */
     /* that the user wants to delete.        */
     found_emp = TRUE;
     db_key = E.RDB$DB_KEY;

/* Pass the dbkey to an external function CALL_OTHER to print   */
/* out the record to which the dbkey points.  Note that using   */
/* an external function is neither necessary nor recommended    */
/* for performing this task.  It is done in this example only   */
/* to show how values are passed between functions in an RDML/C */
/* program.                                                     */
```

(continued on next page)

Example 17–20 (Cont.) Using Structured Programming in RDML/C

```
     call_other (db_key, req_1);
   END_FOR;

  if (found_emp != TRUE)
     printf (" No employee with %s on file\n", employee_id);
  else

     /* Ask user for confirmation that this is the  */
     /* EMPLOYEES record he or she wants to delete. */

     read_string ("Is this the employee you want to delete? (Y,N): ",
                  response, sizeof (response)-1);

COMMIT (TRANSACTION_HANDLE trans_1);


Function CALL_OTHER:

#include <stdio.h>

/* This function is passed the dbkey and transaction handle      */
/* from the DELETE_RECORD function within program C_SAMPLE.RC.   */
/* With this information, the function can find and display the  */
/* employee record associated with an employee_id specified in   */
/* DELETE_RECORD and then return program control to the          */
/* DELETE_RECORD function.               */


/* Because the database was invoked in the main program with     */
/* GLOBAL attributes, refer to it here as EXTERNAL.              */

DATABASE PERS = [EXTERNAL] FILENAME "MF_PERSONNEL";
typedef BASED ON EMPLOYEES.RDB$DB_KEY db_key_type;
globalref int trans_1;

call_other (dbkey, req_1)
db_key_type dbkey;
int req_1;
{

/* The transaction was started in the DELETE_RECORD function,    */
/* so there is no need to start a transaction here.  Use the     */
/* transaction handle to identify this request with the          */
/* transaction started in DELETE_RECORD.  Use the dbkey found    */
/* in the DELETE_RECORD function to locate the correct employee  */
/* record.                                                       */
```

(continued on next page)

Example 17–20 (Cont.) Using Structured Programming in RDML/C

```
    FOR (TRANSACTION_HANDLE trans_1, REQUEST_HANDLE req_1)
       E IN EMPLOYEES WITH E.RDB$DB_KEY = dbkey

           /* Display the EMPLOYEES record.  */

           printf (" \n");
           printf ("Last Name: %s\n", E.LAST_NAME);
           printf ("First Name: %s\n", E.FIRST_NAME);
           printf ("Street: %s\n", E.ADDRESS_DATA_1);
           printf ("Apartment: %s\n", E.ADDRESS_DATA_2);
           printf ("City: %s\n", E.CITY);
           printf ("State: %s\n", E.STATE);
           printf ("Zip Code: %s\n", E.POSTAL_CODE);
           printf ("Sex: %s\n\n", E.SEX);
    END_FOR;
}

/* Return program control to the DELETE_RECORD function.  */
```

17.4.1 Using Handles

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions
- Requests

Information on when and how to use request handles is supplied in Chapter 9. Section 17.4.2 and Section 17.4.4 discuss how to declare handles in an RDML/C program.

17.4.2 Declaring and Initializing Handles

With the exception of the database handle, declaring handles in RDML/C is similar to declaring any other program variable. The declaration and initialization of a database handle is done simply by specifying the handle in the DATABASE statement. You do not declare a database handle in the data declaration portion of your RDML/C program. RDML/C initializes the handle for you. You should not assign a value to a database handle with an assignment statement.

User-specified request and transaction handles must be declared in the data declaration portion of your program. In RDML/C, declare user-specified request and transaction handles as RDML$HANDLE_TYPE and initialize them to zero.

If you want to release the resources associated with a request handle, you can do so by issuing a FINISH statement, or, if you do not want to detach from the database, you can release the request by issuing a call to the RDB$RELEASE_REQUEST procedure with the following statement (where req1 is a user-supplied request handle):

```
if ((RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, &req1) & 1) == 0)
     RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
```

Declare RDB$RELEASE_REQUEST as:

```
extern long RDB$RELEASE_REQUEST();
```

17.4.3 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

17.4.4 Declaring and Initializing Distributed Transaction Identifiers

Declaring distributed transaction identifiers in RDML/C is similar to declaring any other program variable. Distributed transaction identifiers must be declared in the data declaration portion of your C program. Declare a distributed transaction identifier as two longwords and initialize it to zero. You should not assign a value to a distributed transaction identifier with an assignment statement.

17.5 Using Callable RDO

You must use the Callable RDO interface to do either of the following in your RDML application:

- Perform data definition operations within the program.

  The RDML statement set does not include data definition statements. If you want to perform data definition within your RDML/C program, you must use the Callable RDO program interface. For example, your program may define a temporary index on a field to facilitate Rdb/VMS performance during program execution.

- Form dynamic queries.

A dynamic query is one that is not known until run time, and thus is constructed by the application at run time. If you know what the query is before run time, you should use RDML preprocessed statements, because these statements execute significantly faster than Callable RDO statements.

When using Callable RDO, your program communicates with Rdb/VMS using a callable function named RDB$INTERPRET. You call RDB$INTERPRET as you would call a system service. You call RDB$INTERPRET to pass your data manipulation or data definition statements to Rdb/VMS. Declare RDB$INTERPRET as an integer (longword) function. The RDB$INTERPRET function returns a status value that describes the success or failure of the procedure execution. The return status value is a condition value that indicates either success or a unique Rdb/VMS symbolic error code. Your program declares a longword variable to hold the return status value so you can test the success or failure of the call.

Callable RDO program development is explained in detail in Chapter 19.

The C format of the RDB$INTERPRET calling sequence is:

```
ret-stat = RDB$INTERPRET(rdb-statement-desc[,host-var-des,...]);
```

The arguments for the RDB$INTERPRET function are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement-desc

  A pointer to a descriptor that describes the Rdb/VMS statement you are passing to Rdb/VMS. Handle rdb-statement_desc according to the C rules for handling string literals or string variables.

- host-var-desc

  A pointer to a descriptor that describes a host language variable that you pass to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some Rdb/VMS statements may produce unwieldy code in the call to RDB$INTERPRET. Instead, assign the Rdb/VMS statement string literal to a string variable. Then, pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your Rdb/VMS data definition and data manipulation statements from the mechanics of using the Callable RDO interface.

Callable RDO program development is explained in detail in Chapter 19.

The following section discusses the use of the DATABASE statement and the scope of transactions in preprocessed programs that use Callable RDO.

17.5.1 Using the DATABASE Statement with Embedded Callable RDO

You must use a DATABASE statement in your preprocessed program and a separate INVOKE DATABASE statement in the embedded Callable RDO statements. To ensure that RDML invokes the identical database for the preprocessed and Callable RDO portions of the program, use the same database handle in each INVOKE DATABASE statement. Invoke the database:

- In the preprocessed program by using a GLOBAL or EXTERNAL database handle
- In the Callable RDO program by passing the database handle to the RDB$INTERPRET function

For more information on database handles, see the section on handles in Chapter 9.

In Callable RDO, you must pass the database handle to the RDB$INTERPRET function as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both RDML and Callable RDO INVOKE DATABASE statements in the same program module. You may also call a function or subroutine to perform data definition with Callable RDO. In that case, use a preprocessed INVOKE DATABASE statement in the main module and the Callable RDO INVOKE DATABASE statement in the submodule.

For example, in RDM$DEMO:C_SAMPLE.RC, the sample program for C, the database is invoked with the GLOBAL attribute in the main program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME "MF_PERSONNEL" DBKEY SCOPE IS FINISH;
```

This program calls the callable function. This function invokes the database using the RDB$INTERPRET function:

```
strcpy(literal1, "invoke database !val = filename ’mf_personnel’");
command_buf1.dsc$w_length = strlen(literal1);
status = rdb$interpret(&command_buf1, &pers);

if ((status & 1) == 0)
    callable_error(status);
```

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the Callable RDO sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

17.5.2 Embedding Data Definition Statements Using Callable RDO

Data definition statements require a read/write transaction. When an RDML program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You can perform Callable RDO data definition within any active read/write transaction in your preprocessed program. See Section 19.6 for information on using Callable RDO statements and preprocessed statements in a single transaction.

If you call RDB$INTERPRET for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view in Callable RDO and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist, and the preprocessor generates errors for those statements that refer to either the field, relation, or view. You can define indexes, constraints, and any other database elements that are not referred to in the preprocessed code.

You can perform any preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than to rely on implicit START_TRANSACTION declarations.

Example 17–21, from the C function DDL_STMNT, shows how to perform data definition tasks in RDML/C programs.

Example 17–21 Embedding Data Definition Statements in RDML/C

```
ddl_stmnt ()

/* ---------------------------------------------------------- */
/* This function demonstrates how to perform data definition   */
/* tasks from an RDML/C program.  You must use the Callable    */
/* RDO interface, RDB$INTERPRET, to perform data definition    */
/* tasks in preprocessed programs.                             */
/* ---------------------------------------------------------- */

{
char literal [255];             /* RDO command buffer                 */
char literal1 [255];            /* RDO command buffer1                */
char response[80];              /* User's response from read_string() */
long status;                    /* Status returned from RDB$INTERPRET */
long db_handle;                 /* Database handle                    */
int  succeed;

/* Declare descriptors. */

struct dsc$descriptor pers;
$DESCRIPTOR (command_buf, literal);
$DESCRIPTOR (command_buf1, literal1);
/* Set up database handle. */

pers.dsc$a_pointer = &db_handle;
pers.dsc$b_dtype = DSC$K_DTYPE_L;
pers.dsc$b_class = DSC$K_CLASS_S;

status = 0;
/* Prompt user for input.  Ordinarily, it would not be likely that    */
/* you would ask a user to define an index for the database.  This    */
/* example serves only to show you how this type of task can be done  */
/* from within an RDML/C environment.                                 */

while (TRUE)
  {
   response[0] = EOS;
   while (check_response (response, "Y") != 0)
       {
        printf(" \n");
        printf("Please enter the data definition statement to define\n");
        printf("   or delete a temporary index, or type 'exit'\n");
        printf("   on EMPLOYEE_ID, you might enter:\n");
        printf("Define index emp_employee_id for employees. employee_id.\n");
        printf("   end index.\n");
        printf("To delete this index, you might enter:\n");
        read_string("   Delete index emp_employee_id.  :",
                    literal, sizeof (literal)-1);

        if (check_response (literal, "EXIT") == 0)
            return;

        read_string("Did you enter the definition correctly? (Y,N): ",
                    response, sizeof (response)-1);
       }
```

(continued on next page)

Example 17–21 (Cont.) Embedding Data Definition Statements in RDML/C

```
     /* Invoke the database to make it known to Callable RDO. */

     strcpy(literal1, "invoke database !val = filename ’mf_personnel’");
     command_buf1.dsc$w_length = strlen(literal1);
     status = rdb$interpret(&command_buf1, &pers);

     if ((status & 1) == 0)
         callable_error(status);

     /* Start a READ_WRITE transaction. */

     strcpy(literal1, "START_TRANSACTION READ_WRITE");
     command_buf1.dsc$w_length = strlen(literal1);
     status = rdb$interpret (&command_buf1);

     if ((status & 1) == 0)
         callable_error(&status);


/* Pass the data definition statement specified by the user  */
/* to RDB$INTERPRET.                                          */

     command_buf.dsc$w_length = strlen(literal);
     status = rdb$interpret (&command_buf);

     succeed = TRUE;
     if ((status & 1) == 0)
         {
         callable_error(&status);
         succeed = FALSE;
         }
     if (succeed == TRUE)
        {
         printf ("Transaction successful");

         /* Commit */

         strcpy(literal1, "COMMIT");
         command_buf1.dsc$w_length = strlen(literal1);
         status = rdb$interpret (&command_buf1);

         if ((status & 1) == 0)
            callable_error(&status);
         }
     else
        {
        printf ("Transaction failed");

        /* Roll back. */

         strcpy(literal1, "ROLLBACK");
         command_buf1.dsc$w_length = strlen(literal1);
         status = rdb$interpret (&command_buf1);

         if ((status & 1) == 0)
            callable_error(&status);
         }
```

(continued on next page)

Example 17–21 (Cont.) Embedding Data Definition Statements in RDML/C

```
     /* Finish database. */

     strcpy(literal1, "FINISH");
     command_buf1.dsc$w_length = strlen(literal1);
     status = rdb$interpret(&command_buf1, &pers);

     response[0] = EOS;
    }
}/* End ddl_stmnt */
```

17.6 Handling Rdb/VMS Run-Time Errors

Before reading this section, you should be familiar with the information contained in Chapter 10 of this manual. Chapter 10 discusses error handling concepts; this section contains information that, for the most part, is specific to error handling in RDML/C.

This section describes how to detect RDML errors that occur at run time, how to display the accompanying messages, and how to recover from the errors. In most cases, this section assumes that you have debugged the executing program for errors in both RDML and host language statements. This section discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your C user’s guide for such information.

If you choose to combine Callable RDO and RDML, use separate error handling routines for each one. See Chapter 19 for information on handling Callable RDO errors.

17.6.1 Error Handling

RDML/C enables you to detect errors with the ON ERROR clause. If an error occurs in an RDML statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- The ON ERROR clause
- Determining which error has occurred, using the LIB$MATCH_COND run-time library routine
- Error message display, using the SYS$GETMSG and SYS$PUTMSG system services and the LIB$SIGNAL routine

Information on creating user-supplied error messages is contained in Chapter 10.

17.6.2 Detecting Errors Using the ON ERROR Clause

You can use the ON ERROR clause only in preprocessed programs. All RDML statements except the DATABASE and DECLARE_STREAM statements offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you can include one or more host language or Rdb/VMS statements, or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

If you do not use the ON ERROR clause and an Rdb/VMS error occurs, the code generated by calling the RDML$SIGNAL_ERROR error handler passes the error to the VMS Run-Time Library routine, LIB$STOP, which sets the severity level to 4 (FATAL) and forces program termination.

See Chapter 10 for a more complete description of the ON ERROR clause.

The following C code fragment shows the placement of the ON ERROR clause and host language statements within a MODIFY operation:

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
   MODIFY E USING
      ON ERROR
         succeed = FALSE;
         handle_error();
      END_ERROR

      strcpy (E.ADDRESS_DATA_1, street);
      strcpy (E.ADDRESS_DATA_2, address_data);
      strcpy (E.CITY, town);
      strcpy (E.STATE, state);
      strcpy (E.POSTAL_CODE, postal_code);
   END_MODIFY;
END_FOR;
```

17.6.3 Using the RDML General Purpose Error Handler: RDML$SIGNAL_ERROR

The RDML run-time library provides procedures that are used by code generated by RDML. A majority of the routines perform very low-level functions such as building argument lists, internal data transfer, and error handling. None of the present routines is of any real use to application programmers except the general purpose error handler RDML$SIGNAL_ERROR. If an error occurs and you do not use the ON ERROR clause to provide an error handler, RDML uses RDML$SIGNAL_ERROR to call the LIB$STOP routine and your application terminates.

The RDML$SIGNAL_ERROR routine takes a single argument, RDB$MESSAGE_VECTOR. See the next example (in both Pascal and C).

```
READY MINE
   ON ERROR
      RDML$SIGNAL_ERROR (RDB$MESSAGE_VECTOR);
   END_ERROR;
```

Note that in both cases, the ON ERROR clause performs the same error handling task that would be performed by RDML if there were no ON ERROR clause.

If you have decided to use RDML$SIGNAL_ERROR as your error handling routine, there is no need to read the rest of this chapter; it discusses how to use system service routines.

17.6.4 Determining Which Errors Have Occurred

After detecting an error, you want to determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation. You determine which error has occurred by evaluating the symbolic value of the error code.

17.6.4.1 Using Symbolic Error Codes All communication with an Rdb/VMS database is done through procedure calls. In preprocessed programs, RDML/C converts RDML statements to host language calls to Rdb/VMS procedures. Every procedure returns a status value into a program variable that is declared by the preprocessor. The return status value is a longword value that identifies a unique message in the system message file. The return status value may indicate success, in which case data manipulation continues uninterrupted. Or this value may indicate an error, in which case control passes to the error handler.

In RDML/C programs, the preprocessor names this variable RDB$STATUS and declares it to be a longword. The return status value is the same as the value of the second element of a 20-longword array, RDB$MESSAGE_VECTOR. (The RDB$MESSAGE_VECTOR array is the message vector that Rdb/VMS uses to pass information to and from C programs.)

Each error generated by an RDML statement is represented as a symbolic error code. You can use these symbolic error codes to control program logic for specific errors. When the Rdb/VMS ON ERROR clause detects an error, your error handler should do the following:

- Evaluate the symbolic error code either by calling the LIB$MATCH_COND routine or by using a C equality test
- Direct program logic with a C host language statement, such as the switch statement

Although symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. Chapter 10 explains why this is recommended.

17.6.4.2 Declaring Symbolic Error Codes

Rdb/VMS symbolic error codes are longword values. The C declaration is:

```
globalvalue long RDB$_LOCK_CONFLICT;
globalvalue long RDB$_INTEG_FAIL;
```

17.6.4.3 Calling LIB$MATCH_COND

When you want to determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine LIB$MATCH_COND.

You also can evaluate the return status condition code directly with one or more host language statements, without calling the LIB$MATCH_COND routine. Generally, host language statements will use fewer resources than a call to LIB$MATCH_COND. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. You must then link your program again under the new version so the program will detect the correct error codes. The LIB$MATCH_COND routine matches only the condition ID of the return status code and is unaffected by changes in severity levels or facility names.

The LIB$MATCH_COND routine compares the first parameter to each of the remaining parameters in its parameter list. If a match is found, it returns the position in the parameter list of the matching parameter; if no match is found, it returns a zero. You should pass the return status value to the LIB$MATCH_COND routine as the first parameter in the parameter list. In the remaining part of the parameter list, pass the error codes you wish to compare to the return status value. If one of these error codes matches the return status value, the LIB$MATCH_COND routine returns the position of the matching parameter in the order of the remaining part of the parameter list.

For example, suppose you want to determine if RDB$_STREAM_EOF, RDB$_DEADLOCK, or RDB$_NOT_VALID is the return status value. Pass to the LIB$MATCH_COND routine the parameter list that contains the values RDB$STATUS, RDB$_STREAM_EOF, RDB$_DEADLOCK, and RDB$_NOT_VALID. If the value of RDB$STATUS equals the value of RDB$_DEADLOCK, then the LIB$MATCH_COND routine returns a value of 2, because RDB$_DEADLOCK is the second parameter in the remaining part of the parameter list.

Next, use the value that the LIB$MATCH_COND routine returns to determine the path of your error handler’s conditional statement. To continue our example, assume you use a C switch statement as the error handler’s conditional statement. In this example, your switch statement evaluates the value returned by the LIB$MATCH_COND routine, and you execute the second case of the switch statement. Your program performs the statement or statements associated with the case statement. These statements might print a message to the terminal, roll back the transaction, and return program control to a point before the transaction was started. Or they might call a more complex routine to perform these and other actions.

The C format of the call to the LIB$MATCH_COND routine is:

```
err-match = LIB$MATCH_COND(ret-stat, symb-name[,...]);
```

The arguments for this C call are:

- err-match

  A numeric variable that holds the integer that identifies the symbol matched.

- ret-stat

  A pointer to a program variable that holds the return status value (RDB$STATUS) of the last call to the database.

- symb-name

  A pointer to a symbolic error code (or the variable name you have assigned to it) that you want to match against ret-stat. Specify one or more symb-name values, as appropriate. The symbolic error codes are longwords, and are passed by reference.

Declare LIB$MATCH_COND as an external integer function.

Example 17–22 demonstrates the use of the LIB$MATCH_COND routine in a C error handling routine. This error handler could be called from another program that detects errors with an ON ERROR clause and that includes a statement within the ON ERROR . . . END_ERROR block that sets the value of a success flag to FALSE when the ON ERROR clause is executed. This error handler does the following:

- Receives the return status value and the success flag
- Opens a file to record the error messages
- Uses the LIB$MATCH_COND routine to determine which error has occurred
- Uses a switch statement to take different actions depending on which error has occurred
- Sets the success flag to true if corrective error handling could take place
- Closes the file that records the error messages

Example 17–22 Using LIB$MATCH_COND in RDML/C

```
handle_error()
/* ----------------------------------------------------------------*/
/* This function handles run-time errors detected by the ON ERROR   */
/* clause in preprocessed RDML/C programs.                          */
/* ----------------------------------------------------------------*/
{
char msg_string[256];
char string[133];
int error;

FILE open(), fp;
$DESCRIPTOR (msgstr, msg_string);


/* Use LIB$MATCH_COND to determine which of a series of errors */
/* might have occurred.                                        */

error = LIB$MATCH_COND (&RDB$MESSAGE_VECTOR[1],
                        &RDB$_DEADLOCK,
                        &RDB$_LOCK_CONFLICT,
                        &RDB$_NO_DUP,
                        &RDB$_NOT_VALID,
                        &RDB$_INTEG_FAIL,
                        &RDB$_STREAM_EOF,
                        &RDB$_NO_RECORD);

printf (" \n");

/* The switch statement directs program logic to appropriate  */
/* statements to execute depending on the error.              */

switch (error)
     {
      case 0:
         printf("Unexpected error - terminating program\n");
         fp = fopen("error_log", "w");
         error = SYS$GETMSG(RDB$MESSAGE_VECTOR[1], &msgstr.dsc$w_length,
                                                  &msgstr, 0, 0);
         msg_string[msgstr.dsc$w_length] = EOS;
         fputs(msg_string, fp);
         fclose (fp);
         error = LIB$CALLG (&RDB$MESSAGE_VECTOR, LIB$SIGNAL);
         break;
      case 1:
      case 2:
        if (retry <= 4)
          {
          printf("Deadlock or Lock conflict error");
          printf("Others are using the data that you want to access\n");
          error = LIB$WAIT(SECONDS_TO_WAIT);
          }
        else
          printf("Sorry resources are not available, please retry later\n");
        break;
```

(continued on next page)

Example 17–22 (Cont.)   Using LIB$MATCH_COND in RDML/C

```
      case 3:
          printf("Duplicates are not allowed\n");
          SYS$PUTMSG(RDB$MESSAGE_VECTOR);
          break;

      case 4:
          printf("Invalid data\n");
          SYS$PUTMSG(RDB$MESSAGE_VECTOR);
          break;

      case 5:
          printf("Integrity failure");
          SYS$PUTMSG(RDB$MESSAGE_VECTOR);
          break;

      case 6:
          printf("There are no colleges with that code\n");
          break;

      case 7:
          printf("A record entered during this session has been deleted\n");
          break;
     }
}
/* End handle_error */
```

17.6.5 Displaying Error Messages

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and terminate your program, you can call the LIB$SIGNAL routine.
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service.
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service.
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code.

Information on creating user-supplied error messages is contained in Chapter 10.

17.6.5.1 Calling LIB$SIGNAL Call the LIB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and terminate program execution. When you call LIB$SIGNAL with LIB$CALLG, the LIB$SIGNAL routine:

- Receives the signal argument list from the signaling procedure

  This list is made up of the return status value and a set of optional arguments that provide information to condition handlers.

- Copies this signal argument list and uses it to create a signal argument vector

  The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.

- Causes a signal condition which causes the appropriate catchall condition handler to pass the signal argument vector to SYS$PUTMSG

  The SYS$PUTMSG system service calls the SYS$GETMSG system service to retrieve the message from the system error messages, and then formats and displays the error messages on your terminal.

- Resignals the error

  If the error is not fatal, program execution continues. If the error is fatal, the program error handler signals the error to the VMS default condition handler, which terminates program execution.

In C, you can continue program execution after the call to the LIB$SIGNAL routine even when the error is fatal. See Section 17.6.6 for information on how to continue program execution after a fatal error.

17.6.5.2 Methods of Calling LIB$SIGNAL The recommended method of calling LIB$SIGNAL in RDML programs is to pass the message vector, RDB$MESSAGE_VECTOR, and LIB$SIGNAL to the run-time library function, LIB$CALLG.

This method ensures that any Formatted ASCII Output (FAO) arguments that exist in the message vector will be formatted correctly. In addition, this method ensures that any additional error messages that clarify the nature of the program error will be returned to your program. For these reasons, Digital recommends that you always call LIB$SIGNAL with LIB$CALLG.

You can also pass the return status value (RDB$STATUS) to LIB$SIGNAL. However, this method is not recommended. If you pass RDB$STATUS to the LIB$SIGNAL routine and FAO arguments exist in the Rdb/VMS error message, LIB$SIGNAL may be unable to format the Rdb/VMS error message correctly. In this case, your program may terminate abruptly or may provide an incompletely formatted error message.

If your application requires that you call LIB$SIGNAL without LIB$CALLG, be certain that the error message does not contain FAO arguments. Figure 10–1 in Chapter 10 illustrates the format of the message vector.

17.6.5.3 The Format of the LIB$SIGNAL Calling Sequence with RDB$MESSAGE_VECTOR and RDB$STATUS

The C format of the LIB$SIGNAL calling sequence with the message vector (RDB$MESSAGE_VECTOR) is:

```
status = LIB$CALLG(RDB$MESSAGE_VECTOR, LIB$SIGNAL);
```

The LIB$SIGNAL argument is the name of the run-time library routine that will receive RDB$MESSAGE_VECTOR. The LIB$SIGNAL argument is passed by reference in C.

When using the LIB$CALLG routine to pass the message vector, declare LIB$CALLG as:

```
extern long LIB$CALLG();
```

Declare LIB$SIGNAL as:

```
extern long LIB$SIGNAL();
```

An earlier example, Example 17–22, demonstrates how to call LIB$SIGNAL with LIB$CALLG. The C format of the LIB$SIGNAL calling sequence with only the return status value is:

```
LIB$SIGNAL(RDB$STATUS);
```

17.6.5.4 Calling SYS$PUTMSG

Call the SYS$PUTMSG system service when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service writes the error message to the terminal and to the error file designated by the logical name SYS$ERROR. You can define SYS$ERROR at the DCL level to be your program error file when you want the SYS$PUTMSG system service to write an Rdb/VMS error message to it.

The first parameter in the call to the SYS$PUTMSG system service is the message vector RDB$MESSAGE_VECTOR. Figure 10–1 in Chapter 10 illustrates the format of the message vector. The SYS$PUTMSG system service can accept other optional parameters that specify an action routine to receive control during message processing, and the facility name to be used in displaying the message (if you want the facility to be different from the default facility prefix that is associated with the message). The message vector is required; you may omit the optional parameters. See the *VMS System Services Volume* for a complete description of the SYS$PUTMSG system service.

The C format of the SYS$PUTMSG calling sequence is:

```
SYS$PUTMSG(RDB$MESSAGE_VECTOR,NULL,NULL,0);
```

Declare SYS$PUTMSG as an external unsigned function in C.

See an earlier example, Example 17–22, for a demonstration of the use of the SYS$PUTMSG system service.

17.6.5.5 Calling SYS$GETMSG Call the SYS$GETMSG system service when you want to use an error message generated by Rdb/VMS within your program and continue program execution.

The first parameter in the call to the SYS$GETMSG system service is the Rdb/VMS return status value, which is the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program as the second parameter of the call. You must declare a string to receive the message. Your program can then manipulate this string in any way it chooses. Your program can:

- Display the string
- Write the string to a file

You can also evaluate character substrings within the string, but Digital recommends that you do not use this method. The message text may change from one release to the next.

The SYS$GETMSG system service requires a parameter to receive the length of the message string. You may omit the actual parameter, but you must include a comma to signify the argument. The SYS$GETMSG system service accepts other optional parameters that define what is included in the returned message and receives the FAO count of the message. You may omit these parameters; if you do, all components of the message are returned. See the *VMS System Services Volume* for further information on the SYS$GETMSG system service.

The SYS$GETMSG system service does not format the FAO arguments in the error message; instead, it returns the error message with format parameters embedded in it. If your error message contains a view name, for example, the SYS$GETMSG system service will return the message:

```
<View !AC can not be updated>
```

You can call the SYS$FAO system service to format the FAO arguments in the message that the SYS$GETMSG system service returns to your program. However, when the error message contains FAO arguments, you should call SYS$PUTMSG rather than the SYS$GETMSG system service.

The C format of the SYS$GETMSG calling sequence is:

```
ret-stat = SYS$GETMSG(status, msg-len, msg-string,0,0);
```

The arguments of this calling sequence are:

- ret-stat

  A program variable that holds the longword integer that describes the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- status

  A pointer to RDB$STATUS, to a condition code that may be contained in RDB$STATUS, or to one of the RDB$MESSAGE_VECTOR condition codes. This is passed by reference.

- msg-len

  A pointer to a word that holds the number of characters written into msg-string. This is not an optional parameter; if you omit it, you must use a comma in its place. This is passed by reference.

- msg-string

  A pointer to the string variable that holds the returned error message. The maximum length of any message that can be returned is 256 bytes.

Declare SYS$GETMSG as an external integer function.

See an earlier example, Example 17–22, for a demonstration of the use of SYS$GETMSG.

17.6.6 Handling Fatal Errors

In some instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error using the LIB$MATCH_COND routine or one or more host language statements, to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or SYS$GETMSG system service to generate an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

In C, you can also call the LIB$SIGNAL routine to display the error message, but you must use VAXC$ESTABLISH to establish a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

See the *VMS Run-Time Library Routines Volume* for a complete description of the use of LIB$ESTABLISH with LIB$SIGNAL.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary before calling the LIB$SIGNAL routine. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions
- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the LIB$SIGNAL routine without establishing a condition handler, LIB$SIGNAL displays the error message and terminates your program. Perform any cleanup before making the call to LIB$SIGNAL. However, if your cleanup includes any Rdb/VMS statements (such as ROLLBACK), these new calls to the database will change the return status value contained in RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in this case, the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling the LIB$SIGNAL routine without performing any cleanup operations is not recommended.

18 Using the RDML/Pascal Program Environment

This chapter describes how to access an Rdb/VMS database using Pascal and the RDML preprocessor interface.

This chapter presents the following main topics:

- Using Relational Data Manipulation Language (RDML) statements
- Using Rdb/VMS data definition statements
- Error handling in RDML/Pascal

Most examples in this chapter are available on line. The Rdb/VMS installation procedure writes the sample programs to the directory identified by the logical name RDM$DEMO. The file names for these programs are: P_SAMPLE.RPA, P_CALL_OTHER.RPA, and P_ERROR.RPA. The sample program P_SAMPLE.RPA contains most of the procedures referred to in this chapter.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

Note *Before reading this chapter, you should be familiar with the information contained in Chapter 9. The main purpose of this chapter is to provide information and examples specific to VAX Pascal.*

18.1 The RDML/Pascal Preprocessor Interface

When you use the RDML/Pascal preprocessor interface, you simply include Rdb/VMS data manipulation statements directly in your program wherever you need them. When you preprocess the source program, the preprocessor converts the Rdb/VMS data manipulation statements to a series of Pascal calls to Rdb/VMS. At run time, Rdb/VMS executes the calls and returns any retrieved data to the program.

You cannot preprocess a program that attempts to access a non-existent database, unless your database refers to the data dictionary, CDD/Plus, and refers only to the definitions stored there. That is, if you specify a compile-time file name in the DATABASE statement, the database must exist at preprocess time. If you specify a compile-time path name in the DATABASE statement, the path name element must exist in the data dictionary at preprocess time. This is because the preprocessor must be able to validate relation and field definitions in the programs that refer to the database.

Refer to Chapter 11 for information about:

- The RDML preprocessor and its command qualifiers
- Pascal compile qualifiers like /G_FLOATING
- The user-defined option file that is required at the link step

18.2 Embedding RDML Statements in RDML/Pascal Programs

RDML statements are equivalent to the Rdb/VMS data manipulation statements, which are a subset of Relational Database Operator (RDO) utility statements. With these statements you can access a database, update records, retrieve selected records, and handle RDML exception conditions. Refer to the *RDML Reference Manual* for a complete description of the RDML statements.

18.2.1 Converting an RDO Prototype to the RDML/Pascal Program Environment

Once you have created a prototype of your queries in the interactive RDO utility, you are ready to convert these RDO statements to the RDML/Pascal program environment. See Chapter 7 for a discussion of creating an RDO prototype.

Example 18–1 is an RDML/Pascal program based on the RDO prototype examples in Chapter 7.

Example 18–1 Converting an RDO Prototype to RDML/Pascal

```
Procedure Store_cand;

(*--------------------------------------------------------------*)
(* This procedure stores a record in the CANDIDATES relation. In    *)
(* RDML/Pascal, no special work is needed to store VARYING STRING   *)
(* data.                                                            *)
(*--------------------------------------------------------------*)

label test,10;
var
  DECLARE_VARIABLE first_name SAME AS PERS.CANDIDATES.FIRST_NAME;
  DECLARE_VARIABLE last_name SAME AS PERS.CANDIDATES.LAST_NAME;
  DECLARE_VARIABLE middle_init SAME AS PERS.CANDIDATES.MIDDLE_INITIAL;
  DECLARE_VARIABLE status_info SAME AS PERS.CANDIDATES.CANDIDATE_STATUS;

  continue : char;                  (* Continue in module       *)
  succeed : boolean;                (* DML success flag         *)
  transaction_started : boolean;    (* Transaction started flag *)
  err: integer;                     (* Error status             *)

   begin
      succeed := true;
      first_name := '00000';
      continue := 'n';

      (* Prompt user for data to store in the CANDIDATES relation. *)

      while (first_name <> 'exit') or (first_name <> 'EXIT') do
         begin
         while ((continue = 'N') or (continue = 'n')) do
          begin
            write ('Please enter the first name of the candidate or type ');
            writeln ('exit');
            readln  (First_name);
test:       if (first_name = 'exit') or (first_name = 'EXIT')
               then goto 10;
            writeln ('Please enter the middle initial of the candidate');
            readln  (middle_init);
            writeln ('Please enter the last name of the candidate');
            readln  (last_name);
            writeln ('Please enter candidate status information');
            readln  (status_info);
            write ('Have you entered the Candidate information');
            writeln ('correctly?  (Y,N)');
            readln (continue);
            end;
```

(continued on next page)

Example 18–1 (Cont.) Converting an RDO Prototype to RDML/Pascal

```
         (* Start transaction *)

         transaction_started := false;
         lock_error := false;
         retry   := 0;
         while (not transaction_started) and (retry < 5) DO
            begin
              transaction_started := true;
              lock_error := false;
              START_TRANSACTION READ_WRITE RESERVING
                         CANDIDATES FOR SHARED WRITE NOWAIT
                  ON ERROR
                     Handle_error;
                     transaction_started := false;
                     succeed := false;
                  END_ERROR;
               if lock_error then retry := retry + 1;
             end; (* of while *)

          if   transaction_started
          then
               begin
                  succeed := false;
                  retry   := 0;
                  lock_error := false;
                  repeat
                     succeed := true;
                     lock_error := false;

(* Store the values specified by the user in the CANDIDATES       *)
(* relation.  Check for errors and inform the user of the success *)
(* or failure of the STORE operation.                             *)

                     STORE C IN CANDIDATES USING
                         ON ERROR
                            Handle_error;
                            succeed := false;
                          END_ERROR;
                          C.FIRST_NAME := first_name;
                          C.LAST_NAME := last_name;
                          C.MIDDLE_INITIAL := middle_init;
                          C.CANDIDATE_STATUS := status_info;
                     END_STORE;
                  until (succeed) OR ((lock_error) AND (retry > 4)) OR
                        (NOT succeed AND NOT lock_error);

               end; (* if transaction_started *)
```

(continued on next page)

Example 18–1 (Cont.) Converting an RDO Prototype to RDML/Pascal

```
               if succeed
               then
                    begin
                       writeln ('Update operation succeeded');
                       COMMIT;
                    end
               else
                    begin
                       writeln ('Update operation failed');
                       if transaction_started then ROLLBACK;
                     end;

     continue := 'n';
    end;
10: writeln;
end; (* End of store_cand *)
```

The syntax of RDML statements is not identical to the Rdb/VMS DML statements you may be accustomed to using in RDO and RDBPRE. When you incorporate your RDO prototype into your program, you need to remember several differences. In RDML:

- The FOR segmented string statement is used instead of the Rdb/VMS START_SEGMENTED_STRING statement to retrieve segmented strings.
- The STORE segmented string statement is used instead of the Rdb/VMS CREATE_SEGMENTED_STRING statement to store segmented strings.
- The BASED ON clause can be used to declare host language types. RDO has no equivalent statement.
- The DECLARE_VARIABLE clause can be used to declare host language variables. RDO has no equivalent clause.

See Chapter 7 for a full discussion of using prototypes in RDO and for examples of prototype queries.

18.2.1.1 Using Host Language Variables A **host language variable** is a program variable that you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive the values that Rdb/VMS retrieves from the database. Use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following data manipulation statements allow the use of host language variables:

- Any statement that permits the use of an RSE
- DATABASE (you can specify a database handle)
- GET

- READY
- FINISH

When you declare host language variables, follow the Pascal naming rules. Ensure that the data type and size of each host language variable and its corresponding database field are compatible. Refer to Chapter 8 for the lists of equivalent Pascal data types.

You can use the RDML DECLARE_VARIABLE clause to declare host language variables to ensure that the host language variable has the correct data type and size. The DECLARE_VARIABLE clause causes the RDML preprocessor to refer to a database field definition and assign the attributes of that field to the host language variable.

Note that you should not use the DECLARE_VARIABLE clause to declare a variable to hold a segmented string field. The DECLARE_VARIABLE clause does not generate a data type for a segmented string field that is equivalent to the length of the segmented string segment; instead, the DECLARE_VARIABLE clause generates a data type that is equivalent to the logical identifier that points to a segmented string field.

Do not use the DECLARE_VARIABLE clause in RDML/Pascal programs to declare a host language variable for a TEXT field that is referred to in a conditional expression that includes a CONTAINING, MATCHING, or STARTING WITH relational operator. The DECLARE_VARIABLE clause generates a PACKED ARRAY data type for field values of TEXT data type. The CONTAINING, MATCHING, and STARTING WITH relational operators do not execute properly when the comparison value for the conditional expression is a host language variable of PACKED ARRAY data type. When you declare a host language variable that will be compared to a field value of TEXT data type in one of these expressions, use a Pascal expression to declare a VARYING STRING variable to hold the comparison value.

Example 18–2 shows the use of the DECLARE_VARIABLE clause in RDML/Pascal.

Example 18–2 Using DECLARE_VARIABLE to Declare a Host Language Variable in RDML/Pascal

```
DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;
```

For more information on the DECLARE_VARIABLE clause, see Chapter 16 and the *RDML Reference Manual*.

You can use the RDML BASED ON clause to declare host language types, as shown in Example 18–3. The RDML BASED ON clause extracts the data type and size of a field and declares a function with the same attributes. Like the DECLARE_VARIABLE clause, the BASED ON clause should not be used to declare host language variables that are the targets of the CONTAINING, MATCHING, and STARTING WITH conditional expressions in RDML/Pascal programs. The BASED ON clause will also declare types that are based on TEXT fields as a PACKED ARRAY data type.

Example 18–3 Using the BASED ON Clause in RDML/Pascal

```
JOB_CODE_TYPE =   BASED ON JOBS.JOB_CODE;
JOB_TITLE_TYPE =  BASED ON JOBS.JOB_TITLE;
```

You can also declare host language variables by copying database definitions from the data dictionary, CDD/Plus. You can copy relation definitions, which include all the fields within the relation. Copying definitions from the data dictionary ensures consistency and accuracy because you include the database definitions directly in your program's data declaration section. However, you must be careful to copy only those relation and field definitions with data types that are supported by your host language. See Chapter 16 for more information about using CDD/Plus data definitions.

You can use simple and complex Pascal host language variables, such as arrays or records, in an RSE. However, do not use functions or procedures within the RSE. For example, the following Pascal code does not preprocess:

```
(* bad code, won't preprocess! *)
BEGIN
  FOR FIRST 5 E IN EMPLOYEES WITH E.LAST_NAME = SUBSTR(STRING, 1, 24)
  WRITELN (E.LAST_NAME);
  END_FOR;
END.
```

However, you can assign the result of a function to a variable and use the variable within the RSE.

18.2.1.2 Converting DATE Data Type to TEXT DATE data types are stored in Rdb/VMS databases in encoded binary format. To display a date, your program must first retrieve the binary value and convert it into an ASCII string. This is done by using the VMS system service routine, SYS$ASCTIM, to perform the conversion.

See the *VMS System Services Volume* for more information on using the SYS$ASCTIM service.

Note that this Pascal program uses the INHERIT attribute to inherit the SYS$LIBRARY:STARLET.PEN environment. (PEN is an abbreviation for "Pascal Environment.") The STARLET.PEN environment includes the declarations for the VMS system service routines SYS$ASCTIM and SYS$BINTIM.

Example 18–4, a code fragment from the ADD_EMPLOYEES subroutine, demonstrates how to display a date.

Example 18–4 Using ASCTIM System Service Routine in RDML/Pascal

```
FOR E IN EMPLOYEES
  WITH E.RDB$DB_KEY = db_key_array[x]
    ON ERROR
      Handle_error;
    END_ERROR
  writeln (E.FIRST_NAME, E.MIDDLE_INITIAL,
           E.LAST_NAME);
  writeln (E.ADDRESS_DATA_1,E.ADDRESS_DATA_2);
  writeln (E.CITY, E.STATE);
  writeln (E.POSTAL_CODE);

  (* Convert binary date to ascii date *)

  status := $ASCTIM (
            timbuf := ascii_bday,
            timadr := E.BIRTHDAY);
  if (status <> SS$_NORMAL) then
    writeln ('Date conversion failed')
  else
    writeln(ascii_bday);
END_FOR;
```

18.2.1.3 Converting ASCII DATE Strings to Binary Format Use the VMS system service routine, SYS$BINTIM, to convert ASCII DATE strings into a binary representation so the DATE data type fields can be stored in the database.

See the *VMS System Services Volume* for more information on using the SYS$BINTIM system service.

Example 18–5, a code fragment from the ADD_EMPLOYEES subroutine, demonstrates how to use SYS$BINTIM in an RDML/Pascal program.

Example 18–5 Using BINTIM System Service Routine in RDML/Pascal

```
repeat
  writeln ('Please enter the Employee birthday');
  writeln ('In this format: 10-MAY-1986 00:00:00.00');
  readln  (ascii_bday);
  status := $BINTIM (ascii_bday, employee_record.birthday);
  if (status <> SS$_NORMAL)
  then writeln ('Date conversion failed');
until (status = SS$_NORMAL);
```

18.2.2 Using Literals

Use literal values to replace variables in the same way you would in any high-level language. Literal values can be either numeric or character strings. String literals must be quoted in single quotation marks (' ') in Pascal. You may use a literal in any data manipulation statement that accepts a host variable. For example:

```
FOR D IN DEPARTMENTS WITH
   D.DEPARTMENT_CODE = 'ADMN'
     GET
       DEP_NAME = D.DEPARTMENT_NAME;
     END_GET;
END_FOR;
```

18.2.3 Forming Record Streams

In Pascal, and any language that you use to access an Rdb/VMS database, you select the records you are interested in manipulating by gathering records into a stream. You create this stream using the RDML statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

Note that RDML/Pascal does not preprocess an RSE that is immediately followed by a Pascal "WITH record-name" statement. To handle this specific situation and as a general rule, it is good programming practice to place a block within any RDML block-structured statement. Instead of using the structure shown in the following example, use the structure shown in the succeeding example.

Do not use this structure:

```
FOR E IN EMPLOYEES (* RSE *)
  WITH MYREC ...   (* Pascal WITH statement *)
END_FOR;
```

Use this structure instead:

```
FOR E IN EMPLOYEES  (* RSE *)
  Begin            (* Block start *)
    WITH MYREC ... (* Pascal WITH statement *)
  End              (* Block end *)
END_FOR;
```

18.2.4 Retrieving Records

RDML provides you with three statements to retrieve records:

- FOR
- Two START_STREAM statements:
  - Declared START_STREAM
  - Undeclared START_STREAM

18.2.4.1 Using the FOR Statement to Retrieve Records The FOR statement forms a record stream and provides automatic iteration for any RDML and host language statements included within the FOR . . . END_FOR block. The FOR statement always includes an RSE with at least one context variable.

Example 18–6 shows a FOR statement from the DISPLAY_CAND subroutine. It uses the flag “succeed” to determine if the RSE has been satisfied. If a candidate record is found in the relation that matches the values in the host language variables, the succeed flag is set to true. If no record matches the values in the host language variables then the succeed flag remains set to false.

Example 18–6 Using the FOR Statement in RDML/Pascal

```
succeed := FALSE;
FOR C IN CANDIDATES WITH C.FIRST_NAME = first_name
  AND C.MIDDLE_INITIAL = middle_init
  AND C.LAST_NAME = last_name
  writeln (C.FIRST_NAME, ' ',
           C.MIDDLE_INITIAL, ' ', C.LAST_NAME);
  writeln ('has the following status: ', C.CANDIDATE_STATUS);
  succeed := FALSE;
END_FOR;

if (succeed) then
   COMMIT
else
  begin
    writeln ('Candidate not found in database');
    ROLLBACK;
  end;
```

18.2.4.2 Using Declared Streams to Retrieve Records RDML provides two forms of START_STREAM statements, the *declared* and the *undeclared* START_STREAM statements. The declared streams provide all the features of the undeclared streams and more. Most importantly, undeclared streams require that the statements you use to manipulate the stream be enclosed by the START_STREAM and END_STREAM statements in your source program. Declared streams do not impose this restriction. The statements you use to manipulate the stream may appear in any order within your program as long as the DECLARE_STREAM statement appears first and the statements execute in a logical order (START_STREAM, FETCH, GET, END_STREAM).

Digital recommends that all new applications use the declared START_ STREAM statement. For this reason, only the declared START_STREAM statement is discussed in this section. Complete details on the differences between declared and undeclared START_STREAM statements are provided in Chapter 9.

Example 18–7, from the PAIR subroutine, shows the use of the declared START_STREAM statement and the FETCH statement. The example pairs a CANDIDATES record with an EMPLOYEES record at random.

Example 18–7 Using the Declared START_STREAM and FETCH Statements in RDML/Pascal

```
(* Declarations for the subroutine named PAIR            *)
(* Declare two streams: one for the CANDIDATES relation  *)
(* and the other for the EMPLOYEES relation.             *)
   .
   .
   .
DECLARE_STREAM CANDS USING
    CA IN CANDIDATES SORTED BY CA.LAST_NAME;

DECLARE_STREAM EMPS USING
    EM IN EMPLOYEES SORTED BY EM.FIRST_NAME;
   .
   .
   .
(* Set of procedures to control streams in procedure PAIR.        *)
(* Of course, a simple program such as this does not require the  *)
(* use of functions to separate the RDML statements.  It is done  *)
(* here to demonstrate what you can do. Note that the statements  *)
(* do not appear in the order that they will be executed.  This   *)
(* is a feature that declared streams have and undeclared streams *)
(* do not have.                                                   *)

procedure close_emps;
begin
    END_STREAM EMPS;
end;

procedure close_cands;
begin
    END_STREAM CANDS;
end;

procedure read_cands;
begin
    FETCH CANDS
        AT end
         end_of_cands := TRUE;
        END_FETCH;
end;
```

(continued on next page)

Example 18–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDML/Pascal

```
procedure read_emps;
begin
    FETCH EMPS
        AT end
        end_of_emps := TRUE;
    END_FETCH;
end;

procedure open_cands;
begin
    START_STREAM cands;
end;

procedure open_emps;
begin
    START_STREAM emps;
end;

Procedure PAIR;

(* -----------------------------------------------------------------*)
(* This procedure demonstrates the use of the declared START_STREAM *)
(* statement.  The output of this program is merely a random       *)
(* matching of each CANDIDATES record with an EMPLOYEES record. The *)
(* PAIR procedure calls these procedures.                           *)
(* -----------------------------------------------------------------*)


begin
READY PERS;
START_TRANSACTION READ_ONLY;

(* Open both streams and set a flag for the end-of-stream condition *)
(* to false.                                        .               *)

open_cands;
open_emps;

end_of_emps := FALSE;
end_of_cands := FALSE;

(* Fetch a record from the CANDIDATES and EMPLOYEES relations. *)

read_cands;
read_emps;

(* Print the employee and candidate names until the end-of-stream *)
(* condition is met for the stream of CANDIDATES records.        *)
```

(continued on next page)

Example 18–7 (Cont.) Using the Declared START_STREAM and FETCH Statements in RDML/Pascal

```
while not end_of_cands do
  begin
    write (EM.LAST_NAME, ' ', EM.FIRST_NAME);
    write (' ':20);
    writeln (CA.LAST_NAME, ' ', CA.FIRST_NAME);

    read_cands;

    if not end_of_cands then
    read_emps;
   end;

(* Close both streams. *)

close_emps;
close_cands;

COMMIT;
end; (* End PAIR *)
```

18.2.5 Retrieving Segmented Strings

Retrieving segmented strings is a two-step process. First you must retrieve the record that contains the segmented string field, then you must retrieve the individual segments that make up the segmented string field.

You may find it easier to picture a segmented string by referring to Figure 8–1 in Chapter 8.

RDML provides you with the FOR statement with segmented strings to retrieve segmented strings. You must use two streams when processing segmented string streams. Use the first FOR (or START_STREAM) statement to form an outer stream of records, and then use a second FOR statement to form an inner stream of segments. This inner stream identifies the segments contained in the field specified by the first RSE. Use different context variables for the inner and outer streams.

Remember that to retrieve a segmented string, you must begin at the first segment and retrieve segments in the order in which they are stored, that is, sequentially.

Example 18–8 from the DISPLAY_RESUME subroutine:

- Uses a FOR statement to search the database for a record with a value for the EMPLOYEE_ID field that matches the host language variable, employee_id
- Uses a second FOR statement to loop through the segments of the segmented string field for the selected EMPLOYEES record

- Uses a writeln statement to retrieve the individual segments that make up the segmented string
- Displays these values on the terminal

Example 18–8 Using the FOR Statement with Segmented Strings in RDML/Pascal

```
      START_TRANSACTION READ_WRITE RESERVING RESUMES
                        FOR SHARED WRITE;
(* Start an outer FOR loop to retrieve the employee record(s) *)
(* with the specified employee ID.                            *)

        FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id

          (* Start an inner FOR loop to retrieve the segments of *)
          (* the segmented string that make up the employee's    *)
          (* resume.  Display each segment as it is retrieved    *)
          (* from the database.                                  *)

            FOR LINE IN R.RESUME
               writeln (LINE);
            END_FOR;
        END_FOR;
```

18.2.6 Retrieving Field Values

RDML lets you use four methods to retrieve field values as outlined in the following list:

- Use the GET statement to retrieve any value including statistical values and the results of conditional expressions from the database.
- Use the Pascal assignment statement or a Pascal function to retrieve one, several, or all of the fields in a database record and assign those values to one or more host language variables.
- Refer to a field as a parameter of a function.
- Use the write statement to print out database values.

Although you can use an assignment statement to retrieve statistical values and the results of conditional expressions from the database, Digital recommends that you always use the GET statement in these cases. The GET statement lets you perform error checking with the ON ERROR clause, a clause that is not available in statistical functions and conditional expressions. Furthermore, a function call is generated by an assignment statement that is not generated when you use the GET statement. Therefore, the GET statement is more efficient than an assignment statement in the context of statistical and conditional expressions.

Section 18.2.6.1, Section 18.2.6.2, and Section 18.2.6.3 discuss retrieving field and statistical values.

18.2.6.1 Using an Assignment Statement to Retrieve Field Values

When you form a record stream using the FOR statement, you can assign database values to host language variables within the FOR . . . END_FOR block. You can also access database values by using them as parameters to host language functions and parameters.

Example 18–9, from the LIST_RECORD subroutine, demonstrates how to use the Pascal writeln statement to retrieve database values in RDML/Pascal.

Example 18–9 Using an Assignment Statement to Retrieve Field Values in RDML/Pascal

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
  FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
     writeln ('Name is: ', E.FIRST_NAME, E.LAST_NAME);
     writeln ('Degree is: ', D.DEGREE);
     writeln ('Degree field is: ', D.DEGREE_FIELD);
  END_FOR;
   .
   .
   .
END_FOR;
```

When you form a record stream using the START_STREAM statement, you include the FETCH and writeln or assignment statements within the START_STREAM . . . END_STREAM block.

See Example 18–7 for an example of using the FETCH and writeln statements within a START_STREAM . . . END_STREAM block.

18.2.6.2 Using the GET * Statement to Retrieve Records in RDML/Pascal

A special form of the GET statement is the GET * statement, which lets you retrieve database values at the record level rather than the field level. You can retrieve all the fields in a record from a relation with the GET * statement. To use the GET * statement, you must first declare a record structure that contains all the fields in the database relation, with record field names that match the relation field names. You can use the Pascal DICTIONARY statement to create such a record structure, or you can create a Pascal record structure. (See Chapter 16 for more information on copying relation and field definitions from the data dictionary.) The GET * statement in the following example retrieves all the fields in an EMPLOYEES record into the employee_record host language record structure.

```
(* Declare Pascal record structure. *)

 employee_record:
  RECORD
  DECLARE_VARIABLE employee_id       SAME AS PERS.EMPLOYEES.EMPLOYEE_ID;
  DECLARE_VARIABLE last_name         SAME AS PERS.EMPLOYEES.LAST_NAME;
  DECLARE_VARIABLE first_name        SAME AS PERS.EMPLOYEES.FIRST_NAME;
  DECLARE_VARIABLE middle_initial    SAME AS PERS.EMPLOYEES.MIDDLE_INITIAL;
  DECLARE_VARIABLE address_data_1    SAME AS PERS.EMPLOYEES.ADDRESS_DATA_1;
  DECLARE_VARIABLE address_data_2    SAME AS PERS.EMPLOYEES.ADDRESS_DATA_2;
  DECLARE_VARIABLE city              SAME AS PERS.EMPLOYEES.CITY;
  DECLARE_VARIABLE state             SAME AS PERS.EMPLOYEES.STATE;
  DECLARE_VARIABLE postal_code       SAME AS PERS.EMPLOYEES.POSTAL_CODE;
  DECLARE_VARIABLE sex               SAME AS PERS.EMPLOYEES.SEX;
  DECLARE_VARIABLE status_code       SAME AS PERS.EMPLOYEES.STATUS_CODE;
  DECLARE_VARIABLE birthday          SAME AS PERS.EMPLOYEES.BIRTHDAY;
  end;
             .
             .
             .
       FOR FIRST 1 E IN EMPLOYEES
           GET
              employee_record = E.*;
           END_GET;
       END_FOR;
```

18.2.6.3 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly without processing each record in the record stream. The result of a statistical expression is an aggregate, and the data type of the result is often not the same data type as the field on which the statistical expression is performed. See Chapter 8 for information on the data type conversions performed by statistical expressions.

There are two advantages to using a GET rather than an assignment statement. First, the GET statement supports the ON ERROR . . . END_ ERROR clause, which allows you to detect errors that occur during the statistical or Boolean function. Second, using the GET statement results in more efficient code than an assignment statement when it is used with statistical and Boolean functions.

Example 18–10, from the Pascal procedure STATS, uses the COUNT statistical function to find the total number of records in the EMPLOYEES relation.

Example 18–10 Using the GET Statement to Retrieve Statistical Values in RDML/Pascal

```
procedure stats;

(*----------------------------------------------------------------*)
(* This procedure displays the total number of records stored in    *)
(* the EMPLOYEES relation.                                          *)
(*----------------------------------------------------------------*)
var
  err: integer;                              (* Error status *)
  atotal: integer;                           (* total        *)

begin
      START_TRANSACTION READ_ONLY;
(* Use the GET statement with a statistical expression to calculate *)
(* the total number of records in the EMPLOYEES relation.           *)

         write ('The number of employees in the Corporation is:');

         GET
          atotal = (COUNT OF E IN EMPLOYEES);
         END_GET;

         writeln (atotal);
      COMMIT;
end; (* End of stats *)
```

18.2.7 Updating Records Using the STORE, MODIFY, and ERASE Statements

The RDML update statements can only be used within a read/write transaction. (You may, of course, include any valid RDML statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

If you update a record and triggered actions have been defined for the relation containing the record, the update operation (STORE, MODIFY, or ERASE) will have the specified effect on all the relations in the database that have a foreign key relationship with the record you want to update.

If a relation-specific constraint has been defined, your ability to perform update operations may depend on the presence of matching field values in other relations. For more information on relation-specific constraints, see Section 6.6.

Note *You may not use a view to update records if that view refers to more than one relation.*

18.2.7.1 Storing Records You can insert values into one or more fields in one record using a single STORE statement. To store more than one record in a relation, include the STORE statement within a program loop.

Note that RDML may return unpredictable results when a Pascal multipath statement, such as the Pascal case statement, is embedded in an RDML STORE statement. The problem occurs when a field is referred to but not used at run time. This is because RDML assumes that any field mentioned within a STORE . . . END_STORE block is going to be updated.

In the following example, if the program falls through to case 2 at run time, a value will be stored in the FIRST_NAME field even though FIRST_NAME is not referred to in case 2. Upon seeing the field referred to in case 1, RDML sets up a buffer for both the FIRST_NAME and LAST_NAME fields. Because case 2 does not supply data for the FIRST_NAME field, RDML sends to the database whatever happens to be in the buffer for the FIRST_NAME field.

The following code will cause unpredictable results:

```
STORE E IN EMPLOYEES USING
  case i of
   1: begin
        E.LAST_NAME = 'Smith';
        E.FIRST_NAME = 'Andrew';
      end;

   2:
      E.LAST_NAME = 'Jones';
  end;
END_STORE;
```

When different fields are referred to in a multipath statement, the RDML statement should be embedded in the host language multipath statement as shown in the following example:

```
case i of
      1:
           STORE E IN EMPLOYEES USING
              E.LAST_NAME = 'Smith';
              E.FIRST_NAME = 'Andrew';
           END_STORE;
      2:
           STORE E IN EMPLOYEES USING
               E.LAST_NAME = 'Jones';
           END_STORE;
end;
```

Example 18–11, from the STORE_CAND procedure, stores an employee record in the CANDIDATES relation.

Example 18–11 Storing Records in RDML/Pascal

```
transaction_started := false;
lock_error := false;
retry   := 0;

while (not transaction_started) and (retry < 5) DO
  begin
   transaction_started := true;
   lock_error := false;
   START_TRANSACTION READ_WRITE RESERVING
                    CANDIDATES FOR SHARED WRITE NOWAIT
      ON ERROR
        Handle_error;
        transaction_started := false;
        succeed := false;
      END_ERROR;
      if lock_error then retry := retry + 1;
end; (* of while *)
if transaction_started
   then
     begin
      succeed := false;
      retry   := 0;
      lock_error := false;
      repeat
        succeed := true;
        lock_error := false;

(* Store the values specified by the user in the CANDIDATES       *)
(* relation.  Check for errors and inform the user of the success *)
(* or failure of the STORE operation.                             *)

        STORE C IN CANDIDATES USING
          ON ERROR
            Handle_error;
            succeed := false;
          END_ERROR;
          C.FIRST_NAME := first_name;
          C.LAST_NAME := last_name;
          C.MIDDLE_INITIAL := middle_init;
          C.CANDIDATE_STATUS := status_info;
        END_STORE;
      until (succeed) OR ((lock_error) AND (retry > 4)) OR
             (NOT succeed AND NOT lock_error);
```

(continued on next page)

Example 18–11 (Cont.) Storing Records in RDML/Pascal

```
end; (* if transaction_started *)

if succeed
  then
    begin
      writeln ('Update operation succeeded');
      COMMIT;
    end
  else
   begin
     writeln ('Update operation failed');
     if transaction_started then ROLLBACK;
   end;
```

18.2.7.2 Using the STORE * Statement to Store Records A special form of the STORE statement is the STORE * statement, which lets you manipulate database values at the record level rather than the field level. You can store all the fields in a record with the STORE * statement. To use the STORE * statement, you must first declare a record structure that specifies all the fields in the relation definition, with Pascal record field names that match the database field names exactly. Then, put the values you want to store in the database record fields into the Pascal program record and store the entire Pascal record using the STORE * statement. Example 18–12 shows the use of the STORE * statement to store the fields in the employee_record record structure that in turn is stored in the EMPLOYEES relation of the MF_PERSONNEL database.

Example 18–12 Using the STORE * Statement in RDML/Pascal

```
(* Declare a Pascal record structure *)
 employee_record:
  RECORD
  DECLARE_VARIABLE employee_id       SAME AS PERS.EMPLOYEES.EMPLOYEE_ID;
  DECLARE_VARIABLE last_name         SAME AS PERS.EMPLOYEES.LAST_NAME;
  DECLARE_VARIABLE first_name        SAME AS PERS.EMPLOYEES.FIRST_NAME;
  DECLARE_VARIABLE middle_initial    SAME AS PERS.EMPLOYEES.MIDDLE_INITIAL;
  DECLARE_VARIABLE address_data_1    SAME AS PERS.EMPLOYEES.ADDRESS_DATA_1;
  DECLARE_VARIABLE address_data_2    SAME AS PERS.EMPLOYEES.ADDRESS_DATA_2;
  DECLARE_VARIABLE city              SAME AS PERS.EMPLOYEES.CITY;
  DECLARE_VARIABLE state             SAME AS PERS.EMPLOYEES.STATE;
  DECLARE_VARIABLE postal_code       SAME AS PERS.EMPLOYEES.POSTAL_CODE;
  DECLARE_VARIABLE sex               SAME AS PERS.EMPLOYEES.SEX;
  DECLARE_VARIABLE status_code       SAME AS PERS.EMPLOYEES.STATUS_CODE;
  DECLARE_VARIABLE birthday          SAME AS PERS.EMPLOYEES.BIRTHDAY;
  end;
 . . .
(* Assign values to the host language variables. *)
```

(continued on next page)

Example 18–12 (Cont.) Using the STORE * Statement in RDML/Pascal

```
writeln ('Please enter the Employees last name');
readln  (employee_record.last_name);
writeln ('Please enter the Employees first name');
readln  (employee_record.first_name);
 . . .
(* Store these values using the STORE * syntax.  *)

STORE E IN EMPLOYEES USING
  ON ERROR
    Handle_error;
    succeed := false;
  END_ERROR;
  E.*  := employee_record;
 . . .
END_STORE;
```

18.2.7.3 Using the STORE Statement with Segmented Strings to Store Segmented Strings The STORE segmented string statement behaves in a similar manner to the FOR segmented string statement. You must use two streams when you process segmented string streams. Use the first STORE statement to form an outer stream of records, and then use the second STORE statement to form an inner stream of segments. This second STORE statement identifies the segments that are contained in the field specified by the first STORE statement. Use a different context variable in each of the two STORE statements.

Note that the inner STORE statement uses a segmented string variable in place of the context variable, and that the field name is qualified by the context variable specified in the outer STORE statement. Your program must explicitly repeat the inner STORE statement to store individual segments, or provide iteration for an inner STORE loop.

Note *See Section 9.2.6.1.2 for information about defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name with an appropriate value for storing your segmented strings.*

Note *Segmented strings cannot be updated (ERASE, MODIFY, or STORE) as part of a triggered action. For more information, see the DEFINE TRIGGER statement in the* VAX Rdb/VMS RDO and RMU Reference Manual.

Example 18–13, from the STORE_RES subroutine, demonstrates how to store a segmented string in Pascal.

Example 18–13 Storing a Segmented String in RDML/Pascal

```
procedure store_res;
(* This subroutine demonstrates how to store a record with *)
(* a field of data type SEGMENTED STRING.                  *)

label test, 10;
var
  textfile: text;
  my_file : varying [20] of char;
  continue : char;
  err: integer;
  succeed: boolean;

  DEFINE_TYPE employee_id SAME AS RESUMES.EMPLOYEE_ID;
begin
   my_file := 'null';
   employee_id := '00000';
   continue := 'n';
   succeed := true;
   while (Employee_id <> 'exit') or (Employee_id <> 'EXIT') do
      begin
         while ((continue = 'N') or (continue = 'n')) do
            begin

              (* Prompt the user for the employee ID of the        *)
              (* EMPLOYEES record that he or she wants to store.   *)

              write ('Please enter the ID of the employee');
                readln  (Employee_id);
test:           if   (employee_id = 'exit') or (employee_id = 'EXIT')
                then goto 10;

                (* Prompt the user for the file name of the resume *)
                (* to be stored.                                   *)

                writeln ('Please enter file name of the resume');
                readln (my_file);
                writeln ('Have you entered all data correctly? (Y,N)');
                readln (continue);
               end;

           open (textfile, FILE_NAME := my_file,history:=readonly);
           reset (textfile);
           START_TRANSACTION READ_WRITE RESERVING RESUMES
                              FOR SHARED WRITE;
```

(continued on next page)

Example 18–13 (Cont.) Storing a Segmented String in RDML/Pascal

```
(* Use the STORE statement with segmented strings to store the *)
(* record.  The outer STORE statement creates the new RESUMES  *)
(* record.  The inner STORE statement stores the individual    *)
(* segments of the SEGMENTED STRING field.                     *)

           STORE R IN RESUMES USING
              ON ERROR
                  Handle_error;
                  succeed := false;
              END_ERROR;
              R.EMPLOYEE_ID := employee_id;
              While not EOF (textfile)  do
                 begin
                   STORE LINE IN R.RESUME
                      readln (textfile, line);
                   END_STORE;
                 end;
           END_STORE;
           close (textfile);
           if succeed then
               begin
                  writeln ('Resume added ');
                  COMMIT;
               end
            else
               begin
                  writeln ('Update operation failed - resume not added');
                  ROLLBACK;
               end;
           continue := 'n';
        end;
10: writeln;
end; (* end store_res *)
```

18.2.7.4 Modifying Records Using a single MODIFY statement, you can change values in one or more fields of one record in a relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Example 18–14, from the MODIFY_ADDRESS procedure, modifies a record in the EMPLOYEES relation. The values used to modify the record were requested earlier in the program.

Example 18–14 Modifying Records in RDML/Pascal

```
      .
      .
      .
(* Modify the address fields for the specified *)
(* EMPLOYEES record. *)

START_TRANSACTION READ_WRITE
    RESERVING EMPLOYEES FOR SHARED WRITE;

    FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
      MODIFY E USING
         ON ERROR
            Handle_error;
            succeed := false;
         END_ERROR;
         E.ADDRESS_DATA_1 := address_1;
         E.ADDRESS_DATA_2 := address_2;
         E.CITY := city;
         E.STATE := state;
         E.POSTAL_CODE := pcode;
      END_MODIFY;
    END_FOR;
```

18.2.7.4.1 Using the MODIFY * Statement to Modify Records A special form of the MODIFY statement is the MODIFY * statement, which lets you manipulate database values at the record level rather than the field level. You can modify all the fields in a record with the MODIFY * statement. To use the MODIFY * statement, you must first declare a record structure that contains all the fields in the record, with record field names that match the database field names. Then, put the field values you want to replace into the record fields and modify the entire database record using the MODIFY * statement.

Only use the MODIFY * statement if you need to modify every field value in a record. Modifying a field by replacing one value with an identical value, needlessly adds overhead to your program. For example, your program may check constraints on a field value that *you know* is valid because it is the same value that the field presently holds.

Example 18–15 replaces the field values of an employee record in the JOB_HISTORY relation with the values in the job_history host language record structure.

Example 18–15 Using the MODIFY * Statement in RDML/Pascal

```
FOR J IN JOB_HISTORY WITH
  J.EMPLOYEE_ID = EMPLOYEE_ID IN JOB_HISTORY
  AND J.JOB_END MISSING
     MODIFY J USING
        J.* = job_history;
     END_MODIFY
END_FOR
```

18.2.7.4.2 Modifying Segmented Strings The method you use to modify a segmented string involves two RDML statements: the MODIFY statement and the STORE statement with segmented strings. The MODIFY statement selects the records for which you want to modify the segmented string field. An inner STORE statement with segmented strings deletes the existing segmented string and writes over the existing segmented string handle with a new segmented string handle. Note that you cannot modify the individual segments that make up the segmented string, you must replace the entire segmented string.

Example 18–16 demonstrates how to modify a segmented string in RDML/Pascal.

Example 18–16 Modifying Segmented String Fields in RDML/Pascal

```
procedure mod_resume;

(* ------------------------------------------------------------*)
(* This subroutine demonstrates how to modify a field of data   *)
(* type SEGMENTED STRING.                                        *)
(* ------------------------------------------------------------*)

label test, 10;

var
  textfile: text;
  my_file : varying [20] of char;  (* Name of file containing resume *)
  continue : char;                 (* Continue module                *)
  succeed: boolean;                (* Success flag                   *)


  DECLARE_VARIABLE employee_id SAME AS RESUMES.EMPLOYEE_ID;
begin
   my_file := 'null';
   employee_id := '00000';
   continue := 'n';
   succeed := true;

(* Prompt the user to enter employee ID of the RESUMES record he or  *)
(* she wants to modify.                                              *)
```

(continued on next page)

Example 18–16 (Cont.) Modifying Segmented String Fields in RDML/Pascal

```
   while (employee_id <> 'exit') or (employee_id <> 'EXIT') do
      begin
         while ((continue = 'N') or (continue = 'n')) do
            begin
               write ('Please enter the ID of the Employee whose');
               writeln (' resume you want to change or type exit');
               readln  (employee_id);
test:          if   (employee_id = 'exit') or (employee_id = 'EXIT')
                 then goto 10;

(* Prompt user for the file name of the resume that will replace the *)
(* old resume.                                                       *)

               writeln ('To modify a resume, you must supply a new');
               writeln (' file name that contains the new resume');
               writeln ('Please enter file name of new resume');
               readln (my_file);
               writeln ('Have you entered all data correctly? (Y,N)');
               readln (continue);
            end;
         open (textfile, FILE_NAME := my_file,history:=readonly);
         reset (textfile);
         START_TRANSACTION READ_WRITE RESERVING RESUMES
                           FOR SHARED WRITE;
(* Start an outer FOR loop to retrieve the employee record(s) *)
(* with the specified ID.                                     *)

         FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id

(* Use a MODIFY statement to change the value of the *)
(* segmented string field.                           *)

           MODIFY R USING
              ON ERROR
                Handle_error;
                succeed := false;
              END_ERROR;
```

(continued on next page)

Example 18–16 (Cont.) Modifying Segmented String Fields in RDML/Pascal

```
(* Read in the new resume and use a STORE operation to store a new  *)
(* segmented string handle in the RESUMES relation.                 *)

              While not EOF (textfile)  do
                 begin
                    STORE LINE IN R.RESUME
                       readln (textfile, line);
                    END_STORE;
                 end;
           END_MODIFY;
         END_FOR;
         close (textfile);
         if succeed then
             begin
                writeln ('Resume update successful');
                COMMIT;
             end
          else
             begin
                writeln ('Update operation failed - resume not updated');
                ROLLBACK;
             end;
         continue := 'n';
      end;
10: writeln;
end; (* End mod_resume *)
```

18.2.7.5 Erasing Records You can delete one, many, or all the records from a relation using the ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase.

Example 18–17, from the DELETE_RECORD procedure, demonstrates how to ERASE records in RDML/Pascal programs.

Note *The definition of the sample personnel database includes the trigger EMPLOYEE_ID_CASCADE_DELETE, which performs an automatic deletion of records in the relations named in ERASE statements in Example 18–17 (except for RESUMES) when the record with the matching employee ID is deleted from the EMPLOYEES relation. Thus, you would not need to include "cascading deletion" logic in your programs if it were already included in a trigger definition.*

Example 18–17 Erasing Records in RDML/Pascal

```
(* Earlier in the function DELETE_RECORD, an employee record was   *)
(* retrieved to make certain that the user wants to delete this    *)
(* employee’s records. Having made that determination, the program *)
(* will now delete all records associated with that employee. When *)
(* the employee record was retrieved, the database key associated  *)
(* with that record was also retrieved.  It can be used here to    *)
(* quickly locate that employee’s EMPLOYEES record again, so that  *)
(* records for this employee can be erased from all the relations  *)
(* in which he or she has a record.                                *)

START_TRANSACTION READ_WRITE RESERVING EMPLOYEES,
   SALARY_HISTORY, JOB_HISTORY,
   DEGREES, RESUMES FOR SHARED WRITE;
FOR E IN EMPLOYEES WITH E.RDB$DB_KEY = db_key
   ERASE E;
END_FOR;

FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = employee_id
   ERASE JH;
END_FOR;

FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = employee_id
   ERASE SH;
END_FOR;

FOR D IN DEGREES WITH D.EMPLOYEE_ID = employee_id
   ERASE D;
END_FOR;

FOR R IN RESUMES WITH R.EMPLOYEE_ID = employee_id
   ERASE R;
END_FOR;
```

18.3 Controlling the Scope of Database Keys

A **database key** (dbkey) is a logical pointer that has a one-to-one relationship with a record in the database. Each record has a unique dbkey that points to it. You can retrieve this key as though it were a field in a record. For relations, the dbkey is 8 bytes. For views, you can calculate the size by multiplying the number of relations referred to in the view by 8 bytes. If your view refers to only one relation, the dbkey is 8 bytes; if your view refers to two relations, it is 16 bytes, and so on. Once you have retrieved a dbkey, you can use it to retrieve its associated record directly, within the RSE of a FOR or START_STREAM statement.

By default, the scope of a dbkey ends with a COMMIT statement. That is, a dbkey is guaranteed to point to the same record for the life of the transaction in which it is retrieved.

You can override the default scope of COMMIT in your program by specifying in the DATABASE statement that the dbkey scope ends with the FINISH statement.

The following example demonstrates how to specify the dbkey scope in an RDML/Pascal program.

```
DATABASE GLOBAL pers = FILENAME 'MF_PERSONNEL' DBKEY SCOPE IS FINISH;
```

Suggestions on how you can take advantage of the dbkey scope are contained in Section 9.2.7.

18.4 Using Structured Programming

Programs and modules that pass through the RDML preprocessor do not have unlimited freedom in structure. Calls to routines, subprograms, and subroutines require that you pay special attention to the context from which they are called.

Many data manipulation statements, in particular those that use context variables, execute in the context of other data manipulation statements. These statements are:

- FOR
- GET
- DECLARE_STREAM
- START_STREAM
- END_STREAM
- FETCH
- STORE
- MODIFY
- ERASE
- STORE statement with segmented strings
- FOR statement with segmented strings

These individual data manipulation statements each form only part of a complex call to the database. The preprocessor generates one call to the database, using more than one data manipulation statement. For example, a MODIFY statement executes within the context of a FOR or START_STREAM statement. The call to the database can only be made using both the FOR and MODIFY statements. For this reason, the preprocessor requires such data manipulation statements to be lexically sequential, that is, in the order they appear in the program source code.

In structured programming, using program blocks lets you place program statements in an order that promotes program clarity or execution. This order may be entirely different from the order of actual program execution. However, the preprocessor is unaware of the intended run-time order of program block execution. It generates code in the order that data manipulation statements appear in the source code. Keep this in mind when writing your application.

Also keep in mind that a context variable is meaningful only within its scope. In other words, the context variable defined in a FOR statement is meaningless after the END_FOR statement, and a context variable defined in an undeclared START_STREAM statement is meaningless after the END_STREAM statement. However, the context variable defined in a DECLARE_STREAM statement is meaningful throughout the module in which it is issued.

A stream declared with the DECLARE_STREAM statement lets you place the stream manipulation statements in an order that deviates from the order of actual program execution. When you want to use structured programming and you do not need the automatic iteration provided by the FOR statement, use the declared START_STREAM statement.

For more information on the declared and undeclared START_STREAM statement, see Section 9.2.3.2. Data manipulation statements that stand alone as independent calls to the database may appear in any order in the source file. These statements are:

- DATABASE
- READY
- START_TRANSACTION
- GET
- COMMIT
- ROLLBACK
- FINISH
- DECLARE_STREAM

Remember that you must issue the DECLARE_STREAM statement before you can issue a declared START_STREAM statement, and the DATABASE statement must appear in the data declaration section of your program.

Example 18–18, from the DELETE_RECORD and CALL_OTHER subroutines, demonstrates structured programming in a preprocessed Pascal program. The DELETE_RECORD and CALL_OTHER subroutines are in modules that are separately preprocessed and processed. They are linked with the LINK command. The DELETE_RECORD subroutine passes the value of a dbkey to the CALL_OTHER subroutine. This subroutine finds the record associated with the dbkey and displays this record on the terminal. Although it is not necessary to program this query in two modules, it is done here to demonstrate how to pass variables between separately processed modules.

Example 18–18 Using Structured Programming in RDML/Pascal

```
Subroutine DELETE_RECORD:

START_TRANSACTION (TRANSACTION_HANDLE trans_1 ) READ_WRITE;
  found_emp := FALSE;

(* Find the employee record that the user wants to delete *)
(* If an error occurs during the FOR operation, call      *)
(* an error handler.                                      *)

   FOR  (TRANSACTION_HANDLE trans_1)
         E IN EMPLOYEES WITH E.EMPLOYEE_ID = employee_id
       ON ERROR
         Handle_error;
         succeed := false;
       END_ERROR
       found_emp := TRUE;

       (* Get the dbkey of the EMPLOYEES record that  *)
       (* the user wants to delete.                  *)

       db_key := E.RDB$DB_KEY;
   END_FOR;

   if NOT found_emp
   then writeln (' No employee with ', employee_id, 'on file')

(* Pass the dbkey to an external subroutine CALL_OTHER to *)
(* print out the record to which the dbkey points.  Note  *)
(* that using an external subroutine is neither necessary *)
(* nor recommended for performing this task.  It is done  *)
(* in this example only to show how values are passed     *)
(* between subroutines in an RDML/Pascal program.         *)

   else Call_OTHER (db_key,req_1);
   COMMIT (TRANSACTION_HANDLE trans_1);


Subroutine CALL_OTHER:

module transxn (input,output);
(* Because the database was invoked in the main program *)
(* (P_SAMPLE.RPA) with GLOBAL attributes, assign it     *)
(* EXTERNAL scope here.                                 *)

DATABASE pers = [external] FILENAME 'MF_PERSONNEL';
```

(continued on next page)

Example 18–18 (Cont.)   Using Structured Programming in RDML/Pascal

```
(* This subroutine is passed the dbkey and transaction *)
(* handle from the DELETE_RECORD subroutine within     *)
(* program P_SAMPLE.RPA. With this information, the    *)
(* subroutine can find and display the employee record *)
(* associated with an employee_id specified in         *)
(* DELETE_RECORD and then return program control to    *)
(* the DELETE_RECORD subroutine.                       *)

type

  db_key_type = BASED ON EMPLOYEES.RDB$DB_KEY;

var

  trans_1 : [volatile,external] integer;
  req_1 : [volatile,external]integer;
  dbkey : db_key_type;

  [global] procedure call_other(
            key:db_key_type;handle:integer);
begin

(* The transaction was started in the DELETE_RECORD subroutine,  *)
(* so there is no need to start a transaction here.  Use the     *)
(* transaction handle to identify this request with the          *)
(* transaction started in  DELETE_RECORD.  Use the dbkey found   *)
(* in the DELETE_RECORD subroutine to locate the correct         *)
(* employee record.                                              *)

FOR (TRANSACTION_HANDLE trans_1,REQUEST_HANDLE req_1)
   E IN EMPLOYEES WITH E.RDB$DB_KEY = key

      (* Display the EMPLOYEES record. *)

      writeln (E.LAST_NAME);
      writeln (E.FIRST_NAME);
      writeln (E.ADDRESS_DATA_1);
      writeln (E.ADDRESS_DATA_2);
      writeln (E.CITY);
      writeln (E.STATE);
      writeln (E.POSTAL_CODE);
      writeln (E.SEX);
END_FOR;

end;
end.
```

18.4.1 Using Handles

A **handle** is an identifier that you can specify in your program to identify separate instances of the following database objects:

- Databases
- Transactions

- Requests

Information on when and how to use request handles is supplied in Chapter 9. Section 18.4.2 and Section 18.4.4 discuss how to declare handles and identifiers in an RDML/Pascal program.

18.4.2 Declaring and Initializing Handles

With the exception of the database handle, declaring handles in RDML/Pascal is similar to declaring any other program variable. The declaration and initialization of a database handle is done simply by specifying the handle in the DATABASE statement. You do not declare a database handle in the data declaration portion of your RDML/Pascal program. RDML/Pascal initializes the handle for you. You should not assign a value to a database handle with an assignment statement.

User-specified request and transaction handles must be declared in the data declaration portion of your program. In RDML/Pascal, declare user-specified request and transaction handles as RDML$HANDLE_TYPE and initialize them to zero.

If you want to release the resources associated with a request handle, you can do so by issuing a FINISH statement, or, if you do not want to detach from the database, you can release the request by issuing a call to the RDB$RELEASE_REQUEST procedure with the following statement (where req1 is a user-supplied request handle):

```
if not RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, req1) then
         RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
```

You do not need to declare RDB$RELEASE_REQUEST in Pascal programs; it is declared for you in RDMLVPAS.PAS.

18.4.3 Using Distributed Transaction Identifiers

A **distributed transaction identifier** is a variable that uniquely identifies a distributed transaction. When your application coordinates a distributed transaction and explicitly calls DECdtm services, you must pass the distributed transaction identifier to all the databases that are participating in the distributed transaction. You pass the distributed transaction identifier by using the DISTRIBUTED_TRANSACTION keyword with the DISTRIBUTED_TID clause of the START_TRANSACTION statement. The distributed transaction identifier is a readable parameter and is passed by reference.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for information on coordinating a distributed transaction.

18.4.4 Declaring and Initializing Distributed Transaction Identifiers

Declaring distributed transaction identifiers in RDML/Pascal is similar to declaring any other program variable. Distributed transaction identifiers must be declared in the data declaration portion of your Pascal program. Declare a distributed transaction identifier as two longwords and initialize it to zero. You should not assign a value to a distributed transaction identifier with an assignment statement.

18.5 Using Callable RDO

You must use the Callable RDO interface to do either of the following in your RDML application:

- Perform data definition operations within the program.

  The RDML statement set does not include data definition statements. If you want to perform data definition within your RDML/C program, you must use the Callable RDO program interface. For example, your program may define a temporary index on a field to facilitate Rdb/VMS performance during program execution.

- Form dynamic queries

  A dynamic query is one that is not known until run time, and thus is constructed by the application at run time. If you know what the query is before run time, you should use RDML preprocessed statements, because these statements execute significantly faster than Callable RDO statements.

When using Callable RDO, your program communicates with Rdb/VMS using a callable function named RDB$INTERPRET. You call RDB$INTERPRET as you would call a system service. You call RDB$INTERPRET to pass your data manipulation or data definition statements to Rdb/VMS. Declare RDB$INTERPRET as an integer (longword) function. The RDB$INTERPRET function returns a status value that describes the success or failure of the procedure execution. The return status value is a condition value that indicates either success or a unique Rdb/VMS symbolic error code. Your program declares a longword variable to hold the return status value so you can test the success or failure of the call.

Callable RDO program development is explained in detail in Chapter 19.

The Pascal format of the RDB$INTERPRET calling sequence is:

```
ret-stat=RDB$INTERPRET(
          'rdb-statement'[,[%STDESCR][%DESCR]host-var,...]);
```

The arguments for the RDB$INTERPRET calling sequence are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement

  A pointer to a descriptor that describes the Rdb/VMS statement you are passing to Rdb/VMS. Handle rdb-statement according to the Pascal rules for handling string literals or string variables.

- host-var

  A pointer to a descriptor that describes a host language variable that you pass to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some Rdb/VMS statements may produce unwieldy code in the call to RDB$INTERPRET. Instead, assign the Rdb/VMS statement string literal to a string variable. Then, pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your Rdb/VMS data definition and data manipulation statements from the mechanics of using the Callable RDO interface.

Callable RDO program development is explained in detail in Chapter 19.

The following section discusses the use of the INVOKE DATABASE statement and the scope of transactions in preprocessed programs that use Callable RDO.

18.5.1 Using the DATABASE Statement with Embedded Callable RDO

You must use a DATABASE statement in your preprocessed program and a separate INVOKE DATABASE statement in the embedded Callable RDO statements. To ensure that RDML invokes the identical database for the preprocessed and Callable RDO portions of the program, use the same database handle in each INVOKE DATABASE statement. Invoke the database:

- In the preprocessed program by using a GLOBAL or EXTERNAL database handle
- In the Callable RDO program by passing the database handle to RDB$INTERPRET

For more information on database handles, see the section on handles in Chapter 9.

In Callable RDO, you must pass the database handle to the RDB$INTERPRET function as a !VAL parameter. See Chapter 19 for an example of passing database handles in Callable RDO.

You may include both RDML and Callable RDO DATABASE statements in the same program module. You may also call a function or subroutine to perform data definition with Callable RDO. In that case, use a preprocessed INVOKE DATABASE statement in the main module and the Callable RDO INVOKE DATABASE statement in the submodule.

For example, in RDM$DEMO:P_SAMPLE.PAS, the sample program for Pascal, the database is invoked with the GLOBAL attribute in the main program:

```
&RDB&  DATABASE GLOBAL pers = FILENAME ’MF_PERSONNEL’ DBKEY SCOPE IS FINISH;
```

This program calls the callable subroutine. This subroutine invokes the database using the RDB$INTERPRET function:

```
(* Invoke the database to make it known to Callable RDO. *)

     literal1 := ’invoke database !val = filename "mf_personnel"’;
     status := rdb$interpret_integer(literal1, pers);
     if not odd(status) then callable_error(status);
```

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the Callable RDO sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

18.5.2 Embedding Data Definition Statements Using Callable RDO

Data definition statements require a read/write transaction. When an RDML program statement executes, whether it is preprocessed or Callable RDO, Rdb/VMS checks for an active transaction. If there is an active transaction that allows the intended operations, the statement is executed.

You can perform Callable RDO data definition within any active read/write transaction in your preprocessed program. See Section 19.6 for information on using Callable RDO statements and preprocessed statements in a single transaction.

If you call RDB$INTERPRET for data definition, do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Do not define, change, or delete a field, relation, or view in Callable RDO and then refer to it in the preprocessed portion of the program. At preprocess time, the field, relation, or view does not yet exist, and the preprocessor generates errors for those statements that refer to either the field, relation, or view. You can define indexes, constraints, and any other database elements that are not referred to in the preprocessed code.

You can perform any preprocessed data retrieval or update operation within any Callable RDO transaction. You can omit the START_TRANSACTION statement from the preprocessed portion of the program and rely upon the transaction started in the Callable RDO portion. However, it is better practice to begin an explicit transaction whenever possible rather than to rely on implicit START_TRANSACTION declarations.

Example 18–19, from the Pascal subroutine DDL_STMNT, shows how to perform data definition tasks in RDML/Pascal programs.

Example 18–19 Embedding Data Definition Statements in RDML/Pascal

```
procedure ddl_stmnt;

(* -----------------------------------------------------------------*)
(* This subroutine demonstrates how to perform data definition      *)
(* tasks from an RDML/Pascal program.  You must use the Callable    *)
(* RDO interface, RDB$INTERPRET, to perform data definition         *)
(* tasks in preprocessed programs.                                  *)
(* -----------------------------------------------------------------*)


label test, 10;
var
literal : varying [255] of char;  (* RDO command literal buffer        *)
literal1: varying [255] of char;  (* RDO command user buffer           *)
continue: char;                   (* Continue module                   *)
status:integer;                   (* Status returned from RDB$INTERPRET *)
succeed:boolean;                  (* Success of statement              *)
(* Error handler *)

[external] procedure callable_error (var error:integer);
extern;

(* Declare RDB$INTERPRET functions. *)

[ASYNCHRONOUS, EXTERNAL (RDB$INTERPRET)]
    FUNCTION RDB$INTERPRET (
    rdb_statement : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR)
    : integer; EXTERNAL;

[ASYNCHRONOUS, EXTERNAL (RDB$INTERPRET)]
    FUNCTION RDB$INTERPRET_STRING (
    rdb_statement : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
    string_arg : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR)
    : integer; EXTERNAL;
```

(continued on next page)

Example 18–19 (Cont.) Embedding Data Definition Statements in RDML/Pascal

```
[ASYNCHRONOUS, EXTERNAL (RDB$INTERPRET)]
    FUNCTION RDB$INTERPRET_INTEGER (
        rdb_statement : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        %descr  numeric_arg : [list, unsafe] INTEGER)
        : integer; EXTERNAL;

[ASYNCHRONOUS, EXTERNAL (RDB$INTERPRET)]
    FUNCTION RDB$INTERPRET_STRING_INTEGER (
        rdb_statement : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        string_arg : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR;
        %descr  numeric_arg : INTEGER)
        : integer; EXTERNAL;

begin

status := 0;
literal1 := 'goahead';
literal :='goahead';
continue := 'N';
succeed := TRUE;

(* Prompt user for input.  Ordinarily, it would not be likely       *)
(* that you would ask a user to define an index for the             *)
(* database.  This example serves only to show you how this type    *)
(* of task can be done from within a RDML/Pascal environment.       *)


while (literal <> 'exit') or (literal <> 'EXIT') do
  begin
    while ((continue = 'N') or (continue = 'n')) do
      begin
        writeln ('Please enter the data definition statement to define');
        writeln  (' or delete a temporary index, or type "exit"');
        writeln ('For example, to define an index for EMPLOYEES based');
        writeln (' on EMPLOYEE_ID, you might enter: ');
        write   ('define index emp_employee_id for employees. employee_id.');
        writeln ('  end index.');
        write   ('To delete this index, you might enter: ');
        writeln ('delete index emp_employee_id.');
        readln (literal);
test:   if (literal = 'exit') or (literal = 'EXIT')
        then goto 10;
        writeln ('Did you enter the definition correctly? Y,N');
        readln (continue)
      end;

     (* Invoke the database to make it known to Callable RDO. *)

     literal1 := 'invoke database !val = filename "mf_personnel"';
     status := rdb$interpret_integer(literal1, pers);
     if not odd(status) then callable_error(status);

     (* Start a READ_WRITE transaction.                     *)

     status := rdb$interpret  ('START_TRANSACTION READ_WRITE');
     if not odd(status) then callable_error(status);
```

(continued on next page)

Example 18–19 (Cont.) Embedding Data Definition Statements in RDML/Pascal

```
     (* Pass the data definition statement specified by the user *)
     (* to RDB$INTERPRET.                                         *)

     status := rdb$interpret (literal);
     if not odd(status) then
        begin
          callable_error(status);
          succeed := FALSE;
        end;

     if (succeed) then     (* Commit *)
       begin
         writeln ('Transaction successful');
         status := rdb$interpret ('COMMIT');
         if not odd(status) then callable_error(status);
       end

     else
        begin
         writeln ('Transaction failed');  (* Roll back *)
         status := rdb$interpret ('ROLLBACK');
         if not odd(status) then callable_error(status);
        end;
     status := rdb$interpret ('FINISH');   (* Finish database *)
         if not odd(status) then callable_error(status);

     continue := 'n';
    end;
10: writeln;

end; (* End of ddl_stmnt *)
```

18.6 Handling Rdb/VMS Run-Time Errors

Before reading this section, you should be familiar with the information contained in Chapter 10 of this manual. Chapter 10 discusses error handling concepts; this section contains information that, for the most part, is specific to error handling in RDML/Pascal.

This section describes how to detect RDML errors that occur at run time, how to display the accompanying messages, and how to recover from the errors. In most cases, this section assumes that you have debugged the executing program for both RDML and host language statements. This section discusses Rdb/VMS run-time errors only and does not tell you how to handle host language or system run-time errors. Refer to your Pascal user's guide for such information.

If you choose to combine Callable RDO and RDML, use separate error handling routines for each one. See Chapter 19 for information on handling Callable RDO errors.

18.6.1 Error Handling

RDML/Pascal enables you to detect errors with the ON ERROR clause. If an error occurs in an RDML statement, control passes to the ON ERROR clause. Your program must then handle the error.

This section describes:

- The ON ERROR clause
- Determining which error has occurred, using the LIB$MATCH_COND run-time library routine
- Error message display, using the SYS$GETMSG and SYS$PUTMSG system services, user-supplied messages, and the LIB$SIGNAL routine

Information on creating user-supplied error messages is contained in Chapter 10.

18.6.2 Detecting Errors Using the ON ERROR Clause

You can use the ON ERROR clause only in preprocessed programs. All RDML statements except the DATABASE and DECLARE_STREAM statements offer the optional ON ERROR clause. Within the ON ERROR . . . END_ERROR block you can include one or more host language or Rdb/VMS statements, or both. These statements can handle the error directly, but more often they will call an error handler routine that determines the nature of the error and starts appropriate recovery or cleanup procedures.

If you do not use the ON ERROR clause and an Rdb/VMS error occurs, RDML passes the error to the VMS Run-Time Library routine, LIB$STOP, which sets the severity level to 4 (FATAL) and forces program termination.

See Chapter 10 for a more complete description of the ON ERROR clause.

The following Pascal code fragment shows the placement of the ON ERROR clause and host language statements within a MODIFY operation:

```
FOR E IN employees WITH E.EMPLOYEE_ID = employee_id
   MODIFY E USING
         ON ERROR
           success_flag := FALSE;
           Error_handler (RDB$STATUS)
         END_ERROR;

         E.ADDRESS_DATA_1 := address_data_1;
         E.ADDRESS_DATA_2 := address_data_2;
         E.CITY           := city;
         E.STATE          := state;
         E.POSTAL_CODE    := postal_code;
   END_MODIFY;
END_FOR;
```

18.6.3 Using the RDML General Purpose Error Handler: RDML$SIGNAL_ERROR

The RDML run-time library provides procedures that are used by code generated by RDML. A majority of the routines perform very low-level functions such as building argument lists, internal data transfer, and error handling. None of the present routines is of any real use to application programmers except the general purpose error handler RDML$SIGNAL_ERROR. If an error occurs and you do not use the ON ERROR clause to provide an error handler, RDML uses RDML$SIGNAL_ERROR to call the LIB$STOP routine and your application terminates.

The RDML$SIGNAL_ERROR routine takes a single argument, RDB$MESSAGE_VECTOR. For example (in both C and Pascal):

```
READY MINE
   ON ERROR
      RDML$SIGNAL_ERROR (RDB$MESSAGE_VECTOR);
   END_ERROR;
```

Note that in both cases, the ON ERROR clause performs the same error handling task that would be performed by RDML if there were no ON ERROR clause.

If you have decided to use RDML$SIGNAL_ERROR as your error handling routine, there is no need to read the rest of this chapter; it discusses how to use system service routines.

18.6.4 Determining Which Errors Have Occurred

After detecting an error, you want to determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. Recovery might include trying an operation again or writing an error to an error log and continuing to the next operation. You determine which error has occurred by evaluating the symbolic value of the error code.

18.6.4.1 Using Symbolic Error Codes All communication with an Rdb/VMS database is done through procedure calls. In preprocessed programs, RDML/Pascal converts RDML statements to host language calls to Rdb/VMS procedures. Every procedure returns a status value into a program variable that is declared by the preprocessor. The return status value is a longword value that identifies a unique message in the system message file. The return status value may indicate success, in which case data manipulation continues uninterrupted. Or this value may signal an error, in which case control passes to the error handler.

In RDML/Pascal programs, the preprocessor names this variable RDB$STATUS and declares it to be a longword. The return status value is the second element of a 20-longword array, RDB$MESSAGE_VECTOR. (The RDB$MESSAGE_ VECTOR array is the message vector that Rdb/VMS uses to pass information to and from Pascal programs.)

Each error generated by an RDML statement is represented as a symbolic error code. You can use these symbolic error codes to control program logic for specific errors. When the Rdb/VMS ON ERROR clause detects an error, your error handler should:

- Evaluate the symbolic error code either by calling the LIB$MATCH_COND routine or by using a Pascal equality test
- Direct program logic with a Pascal host language statement, such as the EVALUATE statement

Although symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. Chapter 10 explains why this is recommended.

18.6.4.2 Declaring Symbolic Error Codes Rdb/VMS symbolic error codes are longword values. The Pascal declaration is:

```
Var
  RDB$_NO_DUP,
  RDB$_NOT_VALID,
  RDB$_STREAM_EOF : [EXTERNAL] INTEGER;
```

18.6.4.3 Calling LIB$MATCH_COND When you want to determine which of several possible errors has invoked your error handler, you can use the VMS Run-Time Library routine LIB$MATCH_COND.

You also can evaluate the return status condition value directly with one or more host language statements, without calling the LIB$MATCH_COND routine. Generally, host language statements will use fewer resources than a call to LIB$MATCH_COND. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. You must then link your program again under the new version so the program will detect the correct error codes. The LIB$MATCH_COND function matches only the condition ID of the return status code and is unaffected by changes in severity levels or facility names.

The LIB$MATCH_COND routine compares the first parameter to each of the remaining parameters in its parameter list. If a match is found, it returns the position in the parameter list of the matching parameter; if no match is found, it returns a zero. You should pass the return status value to the LIB$MATCH_ COND routine as the first parameter in the parameter list. In the remaining part of the parameter list, pass the error codes you wish to compare to the return status value. If one of these error codes matches the return status value, the LIB$MATCH_COND routine returns the position of the matching parameter in the order of the remaining part of the parameter list.

For example, suppose you want to determine if RDB$_STREAM_EOF, RDB$_DEADLOCK, or RDB$_NOT_VALID is the return status value. Pass to the LIB$MATCH_COND routine the parameter list that contains the values RDB$STATUS, RDB$_STREAM_EOF, RDB$_DEADLOCK, and RDB$_NOT_VALID. If the value of RDB$STATUS equals the value of RDB$_DEADLOCK, then the LIB$MATCH_COND routine returns a value of 2, because RDB$_DEADLOCK is the second parameter in the remaining part of the parameter list.

Next, use the value that the LIB$MATCH_COND routine returns to determine the path of your error handler's conditional statement. To continue our example, assume you use a CASE statement as the error handler's conditional statement. In this example, your CASE statement evaluates the value returned by the LIB$MATCH_COND routine, and your program falls through to the second label of the CASE statement. Your program performs the statement or statements associated with the label statement. These statements might print a message to the terminal, roll back the transaction, and return program control to a point before the transaction was started. Or they might call a more complex routine to perform these and other actions.

The Pascal format of the call to the LIB$MATCH_COND routine is:

```
err-match = LIB$MATCH_COND([%REF]ret-stat, [%REF]symb-name[,...]);
```

The arguments for this Pascal call are:

- err-match

  A numeric variable that holds the integer that identifies the symbol matched.

- ret-stat

  A pointer to a program variable that holds the return status value (RDB$STATUS) of the last call to the database.

- symb-name

  A pointer to a symbolic error code (or the variable name you have assigned to it) that you want to match against ret-stat. Specify one or more symb-name values, as appropriate. The symbolic error codes are longwords, and are passed by reference.

Declare LIB$MATCH_COND as an external integer function.

Example 18–20 demonstrates the use of LIB$MATCH_COND in a Pascal error handling routine. This error handler could be called from another program that detects errors with an ON ERROR clause and that includes a statement within the ON ERROR . . . END_ERROR block that sets the value of a success flag to FALSE when the ON ERROR clause is executed. This error handler does the following:

- Receives the return status value and the success flag
- Opens a file to record the error messages
- Uses the LIB$MATCH_COND routine to determine which error has occurred
- Uses a CASE statement to take different actions depending on which error has occurred
- Sets the success flag to true if corrective error handling could take place
- Closes the file that records the error messages

Example 18–20 Using LIB$MATCH_COND in RDML/Pascal

```
PROCEDURE Handle_error;

(* Error handler for preprocessed portion of program *)
CONST
    seconds_to_wait = 5;
TYPE
  string_type = packed array [1..128] of char;
  $ubyte = [byte] 0..255;
  $uword = [word] 0..65535;
  starlet$$typ9 = [unsafe] array [1..4] of $ubyte;
  vector_type = ARRAY [1..20] OF INTEGER;
VAR
  (* Declare Rdb/VMS symbolic error codes. *)
  RDB$_DEADLOCK,
  RDB$_LOCK_CONFLICT,
  RDB$_INTEG_FAIL,
  RDB$_NO_DUP,
  RDB$_NOT_VALID,
  RDB$_STREAM_EOF,
  RDB$_NO_RECORD:[value,EXTERNAL]INTEGER;

  msg_string:string_type;
  msg_len :integer;
  error: integer;
  string: varying [133] of char;
  return_status: integer;

(* Declare system services to handle errors. *)

[ASYNCHRONOUS, EXTERNAL] FUNCTION LIB$MATCH_COND
(ret_status:INTEGER; sym_name:[LIST]INTEGER):INTEGER;EXTERNAL;

[EXTERNAL] FUNCTION LIB$WAIT(seconds:SINGLE):INTEGER;EXTERNAL;
```

(continued on next page)

Example 18–20 (Cont.) Using LIB$MATCH_COND in RDML/Pascal

```
[ASYNCHRONOUS,EXTERNAL(SYS$GETMSG)] FUNCTION GETMSG (
        %IMMED msgid : UNSIGNED;
        VAR msglen : [VOLATILE]$UWORD;
        VAR bufadr : [CLASS_S] PACKED ARRAY [$l3..$u3:INTEGER] OF CHAR;
        %IMMED flags : UNSIGNED := %IMMED 15;
        VAR outadr : [VOLATILE]STARLET$$TYP9 := %IMMED 0) : integer; external;

[ASYNCHRONOUS,EXTERNAL(LIB$signal)] FUNCTION LIB_SIG(
%IMMED ret_status : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS,EXTERNAL] FUNCTION LIB$CALLG
(stat_vector : vector_type; proced:UNSIGNED) : INTEGER; EXTERNAL;

PROCEDURE SYS$putmsg (stat_vector:vector_type);EXTERNAL;
PROCEDURE RDB$signal ; EXTERNAL;
begin

  (* Use LIB$MATCH_COND to determine which error has occurred. *)

  error := LIB$MATCH_COND (RDB$MESSAGE_VECTOR[2],
                            IADDRESS (RDB$_DEADLOCK),
                            IADDRESS (RDB$_LOCK_CONFLICT),
                            IADDRESS (RDB$_NO_DUP),
                            IADDRESS (RDB$_NOT_VALID),
                            IADDRESS (RDB$_INTEG_FAIL),
                            IADDRESS (RDB$_STREAM_EOF),
                            IADDRESS (RDB$_NO_RECORD));
  writeln ;
  (* The CASE statement directs program logic depending on the *)
  (* type of error that occurs. *)

  CASE error OF
   0:
    begin
      writeln ('Unexpected error - terminating program');
      open (errorfile,FILE_NAME := 'error_log');
      EXTend (errorfile);
      error := GETMSG(%IMMED RDB$MESSAGE_VECTOR[2],%REF msg_len,
                        %stdescr msg_string,%IMMED 0, %IMMED 0);
      writeln (errorfile,msg_string:msg_len);
      close (errorfile);
      error := LIB$CALLG (%REF RDB$MESSAGE_VECTOR, %IMMED LIB_SIG);
    end;
   1,2 :
    begin
      lock_error := true;
      if   retry <= 4
      then
         begin
           writeln ('Deadlock or Lock conflict error');
           writeln ('Others are using the data that you want to access');
           return_status := LIB$WAIT(seconds_to_wait);
           writeln ('Trying to access the data again');
          end
      else
         writeln ('Sorry, resources are not available, please retry later');
    end;
```

(continued on next page)

Example 18–20 (Cont.) Using LIB$MATCH_COND in RDML/Pascal

```
   3:
    begin
       writeln ('Duplicates are not allowed');
       (* Display error message to see what index violated duplicate clause.*)
       SYS$putmsg (RDB$MESSAGE_VECTOR);
    end;
   4:
    begin
       writeln ('Invalid data');
       (* Display error message to see what data was invalid. *)
       SYS$putmsg (RDB$MESSAGE_VECTOR);
    end;
   5:
    begin
       writeln ('Integrity failure');
       (* Display error message to see what data was invalid. *)
       SYS$putmsg (RDB$MESSAGE_VECTOR);
    end;
   6:
    begin
       string := ('There are no colleges with that code');
       writeln (string);
    end;
   7:  writeln ('A record entered during this session has been deleted');
  end; (* case *)
end;
```

18.6.5 Displaying Error Messages

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and terminate your program, you can call the LIB$SIGNAL routine.
- Display an error message generated by Rdb/VMS and continue program execution, you can call the SYS$PUTMSG system service.
- Use an error message generated by Rdb/VMS within your program and continue program execution, you can call the SYS$GETMSG system service.
- Display user-supplied error messages, you can call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code.

Information on creating user-supplied error messages is contained in Chapter 10.

Note *VAX Pascal kits include a file in SYS$LIBRARY, named STARLET.PEN. This file contains Pascal declarations for system service routines. If this file is installed on your system, you do not have to declare system services in your Pascal programs. To make this file known to your program,* *place the following statement before your program statement: [INHERIT 'SYS$LIBRARY:STARLET'].*

18.6.5.1 Calling LIB$SIGNAL Call the LIB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and terminate program execution. When you call LIB$SIGNAL with LIB$CALLG, the LIB$SIGNAL routine:

- Receives the signal argument list from the signaling procedure

  This list is made up of the return status value and a set of optional arguments that provide information to condition handlers.
- Copies this signal argument list and uses it to create a signal argument vector

  The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.
- Causes a signal condition which causes the appropriate catchall condition handler to pass the signal argument vector to SYS$PUTMSG

  The SYS$PUTMSG system service calls the SYS$GETMSG system service to retrieve the message from the system error messages, and then formats and displays the error messages on your terminal.
- Resignals the error

  If the error is not fatal, program execution continues. If the error is fatal, the program error handler signals the error to the VMS default condition handler, which terminates program execution.

In Pascal, you can continue program execution after the call to the LIB$SIGNAL routine even when the error is fatal. See Section 18.6.6 for information on how to continue program execution after a fatal error.

18.6.5.2 Methods of Calling LIB$SIGNAL The recommended method of calling LIB$SIGNAL in RDML programs is to pass the message vector, RDB$MESSAGE_VECTOR, and LIB$SIGNAL to the run-time library function, LIB$CALLG.

This method ensures that any Formatted ASCII Output (FAO) arguments that exist in the message vector will be formatted correctly. In addition, this method ensures that any additional error messages that clarify the nature of the program error will be returned to your program. For these reasons, Digital recommends that you always call LIB$SIGNAL with LIB$CALLG.

You can also pass the return status value (RDB$STATUS) to LIB$SIGNAL. However, this method is not recommended. If you pass RDB$STATUS to the LIB$SIGNAL routine and FAO arguments exist in the Rdb/VMS error message, LIB$SIGNAL may be unable to format the Rdb/VMS error message correctly. In this case, your program may terminate abruptly or may provide an incompletely formatted error message.

If your application requires that you call LIB$SIGNAL without LIB$CALLG, be certain that the error message does not contain FAO arguments. Figure 10–1 in Chapter 10 illustrates the format of the signal argument vector.

18.6.5.3 The Format of the LIB$SIGNAL Calling Sequence with RDB$MESSAGE_VECTOR and RDB$STATUS The Pascal format of the LIB$SIGNAL calling sequence with the message vector (RDB$MESSAGE_VECTOR) is:

```
ret-stat = LIB$CALLG[%REF]RDB$MESSAGE_VECTOR, %IMMED LIB_SIG);
 where LIB_SIG is declared as:
[ASYNCHRONOUS, EXTERNAL(LIB$SIGNAL)]FUNCTION LIB_SIG(
%IMMED RET_STATUS INTEGER) : INTEGER;
EXTERN;
```

The LIB$SIGNAL argument is the name of the run-time library routine that will receive RDB$MESSAGE_VECTOR. The LIB$SIGNAL argument is passed by reference in Pascal.

When using the LIB$CALLG routine to pass the message vector, do not declare LIB$CALLG. The Pascal preprocessor includes in your program the RDBVPAS.PAS file, which declares LIB$CALLG.

When using LIB$CALLG, you must declare LIB$SIGNAL as:

```
[ASYNCHRONOUS, EXTERNAL (LIB$SIGNAL)]FUNCTION LIB_SIG(
%IMMED RET_STATUS INTEGER) : INTEGER;
EXTERN;
```

An earlier example, Example 18–20, demonstrates how to call LIB$SIGNAL with LIB$CALLG.

The Pascal format of the LIB$SIGNAL calling sequence with the return status value is:

```
LIB$SIGNAL([%IMMED]RDB$STATUS]);
```

Declare LIB$SIGNAL as an external integer function.

18.6.5.4 Calling SYS$PUTMSG Call the SYS$PUTMSG system service when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service writes the error message to the terminal and to the error file designated by the logical name SYS$ERROR. You can define SYS$ERROR at the DCL level to be your program error file when you want the SYS$PUTMSG system service to write an Rdb/VMS error message to it.

The first parameter in the call to the SYS$PUTMSG service is the message vector RDB$MESSAGE_VECTOR. Figure 10–1 in Chapter 10 illustrates the format of the message vector. The SYS$PUTMSG system service can accept other optional parameters that specify an action routine to receive control during message processing, and the facility name to be used in displaying the message (if you want the facility to be different from the default facility prefix that is associated with the message). The message vector is required; you may omit the optional parameters. See the *VMS System Services Volume* for a complete description of the SYS$PUTMSG system service.

The Pascal format of the SYS$PUTMSG calling sequence is:

```
SYS$PUTMSG([%REF]RDB$MESSAGE_VECTOR);
```

Declare SYS$PUTMSG as an external integer function in Pascal.

See an earlier example, Example 18–20, for a demonstration of the use of the SYS$PUTMSG system service.

18.6.5.5 Calling SYS$GETMSG Call the SYS$GETMSG system service when you want to use an error message generated by Rdb/VMS within your program and continue program execution.

The first parameter in the call to the SYS$GETMSG system service is the Rdb/VMS return status value, which is the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program as the second parameter of the call. You must declare a string to receive the message. Your program can then manipulate this string in any way it chooses. Your program can:

- Display the string
- Write the string to a file

You can also evaluate character substrings within the string, but Digital recommends that you do not use this method. The message text may change from one release to the next.

The SYS$GETMSG system service requires a parameter to receive the length of the message string. You may omit the actual parameter, but you must include a comma to signify the argument. The SYS$GETMSG system service accepts other optional parameters that define what is included in the returned message and receive the FAO count of the message. You may omit these parameters; if you do, all components of the message are returned. See the *VMS System Services Volume* for further information on the SYS$GETMSG system service.

The SYS$GETMSG system service does not format the FAO arguments in the error message; instead, it returns the error message with format parameters embedded in it. If your error message contains a view name, for example, the SYS$GETMSG system service will return the message:

```
<View !AC can not be updated>
```

You can call the SYS$FAO system service to format the FAO arguments in the message that the SYS$GETMSG system service returns to your program. However, when the error message contains FAO arguments, you should call SYS$PUTMSG rather than the SYS$GETMSG system service.

The Pascal format of the SYS$GETMSG calling sequence is:

```
ret-stat = SYS$GETMSG(
 [%IMMED]RDB$STATUS, [%IMMED msg-len],[%STDESCR]msg-string,
 %IMMED 0,%IMMED 0);
```

The arguments of this callings sequence are:

- ret-stat

  A program variable that holds the longword integer that describes the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- status

  A pointer to RDB$STATUS, to a condition code that may be contained in RDB$STATUS, or to one of the RDB$MESSAGE_VECTOR condition codes. This is passed by reference.

- msg-len

  A pointer to a word that holds the number of characters written into msg-string. This is not an optional parameter; if you omit it, you must use a comma in its place. This is passed by reference.

- msg-string

  A pointer to the string variable that holds the returned error message. The maximum length of any message that can be returned is 256 bytes.

Declare SYS$GETMSG as an external integer function.

See an earlier example, Example 18–20, for a demonstration of the use of SYS$GETMSG.

18.6.6 Handling Fatal Errors

In some instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator, or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program might:

- Evaluate the error using the LIB$MATCH_COND routine or one or more host language statements, to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or SYS$GETMSG system service to generate an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

In Pascal, you can also call the LIB$SIGNAL routine to display the error message, but you must use LIB$ESTABLISH, or the Pascal function ESTABLISH to create a condition handler that will permit your program to continue after the call to LIB$SIGNAL.

See the *VMS Run-Time Library Routines Volume* for a complete description of the use of LIB$ESTABLISH with LIB$SIGNAL.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary before calling the LIB$SIGNAL routine. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions
- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the LIB$SIGNAL routine without establishing a condition handler, LIB$SIGNAL displays the error message and terminates your program. Perform any cleanup before making the call to LIB$SIGNAL. However, if your cleanup includes any Rdb/VMS statements (such as ROLLBACK), these new calls to the database will change the return status value contained in

RDB$STATUS. Therefore, save the return status value of the fatal error in a variable before executing other Rdb/VMS statements, then pass the original return status value to the LIB$SIGNAL routine.

You can call the LIB$SIGNAL routine without performing any Rdb/VMS cleanup operations; in that case, the database monitor will roll back the transaction and perform the necessary database cleanup. However, calling the LIB$SIGNAL routine without performing any cleanup operations is not recommended.

19 Using the Callable RDO Program Environment

This chapter describes how to access an Rdb/VMS database using the Callable RDO program interface. You must use this interface when Rdb/VMS does not support a preprocessor for your program language. You may also use this interface when you want to perform Rdb/VMS data definition tasks or dynamic data manipulation tasks in BASIC, C, COBOL, FORTRAN, or Pascal programs. The information in this chapter is applicable to any VAX program language that supports the VAX Procedure Calling Standard. This chapter presents these main topics:

- Callable RDO program interface
- Converting queries to the program environment
- Using RDB$INTERPRET
- Using data manipulation statements
- Using data definition statements
- Mixing preprocessed and Callable RDO statements in a single transaction
- Handling Rdb/VMS errors

Most examples in this chapter are taken from sample programs included in the online RDM$DEMO directory; these programs work with the sample personnel database. The file names are PLI_SAMPLE.PLI and PLI_CALL_OTHER.PLI for the examples written in PL/I. The file name is DEPTFOR.FOR for the sample program written in FORTRAN.

Note that many of these examples do not perform all the error handling tasks that an application program should perform. Your program, of course, should anticipate as many errors as possible. Only a few error handling tasks have been included in the example programs in order to emphasize only the specific operation being discussed.

For examples of BASIC, COBOL, FORTRAN, Pascal, and C programs that use Callable RDO, refer to the language-specific chapters.

19.1 The Callable RDO Program Interface

When you use the Callable RDO program interface, your program communicates with Rdb/VMS using a callable function, RDB$INTERPRET. Unlike preprocessor interfaces, the Callable RDO interface performs in an interpretive manner.

The Rdb/VMS statements you use in your program are string literals. When the program executes, the statements are passed to Rdb/VMS in the calls to RDB$INTERPRET. The interactive Rdb/VMS interface, RDO, then interprets and executes them.

You call the RDB$INTERPRET function as you would call a VMS Run-Time Library routine. In the calling sequence, you pass both Rdb/VMS statements and host language variables that cause values from the database to be retrieved or updated. The call to RDB$INTERPRET returns a status value that indicates the success or failure of the statements. If the call was successful, RDB$INTERPRET also returns retrieved database values to the appropriate program variables.

Callable RDO is significantly slower than preprocessed Rdb/VMS data manipulation statements, because the Callable RDO statements must be interpreted at run time. You should use Callable RDO only when any of the following applies:

- An RDBPRE, RDML, or SQL preprocessor does not exist for your host language.

  See the *VAX Rdb/VMS Guide to Using SQL* for more information on the SQL preprocessors.

- Your program must perform data definition tasks and you cannot use the SQL interface.

  The Rdb/VMS preprocessors, RDBPRE and RDML, do not support data definition tasks.

- Your program must perform dynamic data manipulation tasks.

  A dynamic data manipulation task is one that is not coded into the application program. That is, you do not know what the query is until run time.

19.1.1 Using Rdb/VMS Data Manipulation Statements

The RDO data manipulation statements are a subset of the RDO statement set. With RDO data manipulation statements, you can access a database, update records, and retrieve values from selected records in the database.

You cannot use the FOR statement in Callable RDO programs. The FOR statement provides automatic iteration through all the records in the record stream. A call to RDB$INTERPRET can, at most, update or retrieve one record at a time. Thus, the FOR loop will fail after the first iteration in a call to RDB$INTERPRET. Instead, use one of the START_STREAM statements and the FETCH statement to retrieve data and repeat the calls within your program structure.

Make sure that you do not issue preprocessed data manipulation statements that rely on metadata defined in the interpreted sections of the same program. The preprocessor will not be able to refer to metadata that has not yet been defined.

Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of all the Rdb/VMS data manipulation statements.

19.1.2 Using Rdb/VMS Data Definition Statements

The Rdb/VMS data definition statements are a subset of the RDO statement set. With Rdb/VMS data definition statements, you can access a database and define, change, or delete the following database elements:

- Fields
- Relations
- Views
- Indexes
- Constraints
- Triggers
- Storage maps

Do not attempt to use database or transaction handles in your data definition statements. Rdb/VMS does not support the use of database or transaction handles in data definition statements.

Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* for a complete description of the Rdb/VMS data definition statements.

19.2 Converting Queries to the Program Environment

Once you have a prototype of your queries using interactive RDO, you are ready to convert these statements to the host language program environment. See Chapter 7 for a full discussion of developing prototype applications in RDO and for examples of prototype queries.

When you incorporate your prototype RDO statements into a program, however, you need to consider these areas of difference:

- Use of host language variables
- Differences in syntax
  - Using the GET statement instead of the PRINT statement
  - Using the START_STREAM and END_STREAM statements instead of the FOR loop
  - Nesting FETCH and GET operations within a host language loop
- Scope of the database attach:

  You cannot invoke a database in Callable RDO programs with the GLOBAL or EXTERNAL scope.

Subsequent sections discuss these differences in detail.

19.3 Using RDB$INTERPRET

You call the RDB$INTERPRET function to pass your data manipulation or data definition statement to Rdb/VMS. Declare RDB$INTERPRET as an external integer (longword) function. Refer to your programming language reference manual for further instructions about declaring such functions.

The RDB$INTERPRET function returns a status value that indicates the success or failure of the function's execution. The return status value indicates either success or a specific Rdb/VMS error code.

Your program declares a longword variable to hold the return status value so you can test for the success or failure of the call. (Refer to Chapter 10 and Section 19.7 for further information on handling Rdb/VMS run-time exception conditions.)

For example, the PL/I format of the RDB$INTERPRET calling sequence is:

```
ret-stat = RDB$INTERPRET(DESCRIPTOR(rdo-statement)
           [,DESCRIPTOR(host-var),...]);
```

The arguments for the RDB$INTERPRET function are:

- ret-stat

  A program variable that holds the longword integer that indicates the success or failure of the call. Your program tests the value of ret-stat and optionally branches to a routine for handling exception conditions.

- rdb-statement

  The RDO statement you are passing to Rdb/VMS. Handle rdb-statement according to your host language's rules for handling string literals or string variables.

- host-var

  A host language variable you are passing to Rdb/VMS as part of a data manipulation statement. You do not include host language variables within the Rdb/VMS statement string literal, but pass them, *in order*, after the string literal.

The RDB$INTERPRET function requires all parameters (the Rdb/VMS statement and host language variables) to be passed *by descriptor*. You must include a by-descriptor passing mechanism when your language's default passing mechanism for host language variables is not by descriptor. Refer to your programming language reference manual for the specific format of the passing mechanism.

You can include rdb-statement in the calling sequence directly as a string literal. However, the length of some RDO statements may produce unwieldy code in the call to RDB$INTERPRET. Instead, assign the RDO statement string to a string variable. Then pass the string variable in the calling sequence. Assigning Rdb/VMS statements to a string variable lets you separate your data manipulation statements from the mechanics of using the RDB$INTERPRET function.

Note *Callable RDO interprets a hyphen between two variables or strings (with no intervening spaces) as an underscore. For example, A-B is interpreted as A_B. If you want a hyphen to be interpreted as a hyphen, leave a blank space on each side of it; for example, A - B.*

Example 19–1, from the sample program PLI_SAMPLE.PLI, shows a call to RDB$INTERPRET. This program assigns the string that contains the START_ STREAM statement to the PL/I host language variable RDB_COMMAND, and then passes this value to the RDB$INTERPRET function to perform the data manipulation task.

Example 19–1 Using RDB$INTERPRET in PL/I

```
RDB_COMMAND = 'START_STREAM ES USING E IN '
              !! 'EMPLOYEES SORTED BY E.LAST_NAME';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;
```

19.4 Using Rdb/VMS Data Manipulation Statements

Depending upon your particular application, you need to perform some or all of the following data manipulation functions:

- Pass values between your program and Rdb/VMS.
- Access one or more databases using the INVOKE DATABASE statement.
- Start a transaction using the START_TRANSACTION statement.
- Form a record stream using the START_STREAM and END_STREAM statements.
- Form a segmented string stream using the START_SEGMENTED_STRING statement.
- Update or retrieve records using the STORE, MODIFY, ERASE, or GET statements.
- Roll back the transaction using the ROLLBACK statement.
- Commit the updates using the COMMIT statement.

19.4.1 Declaring Host Language Variables

A **host language variable** is a program variable you use to communicate with Rdb/VMS. A host language variable can contain the values that update the database; it can also receive values that Rdb/VMS retrieves from the database. Use host language variables as value expressions in data manipulation statements, as well as for any other program function. The following data manipulation statements allow the use of host language variables:

- Any statement that permits the use of an RSE
- GET
- DATABASE
- READY
- FINISH

When you declare host language variables, follow the naming rules for your language. Ensure that host language variable data types and sizes are compatible with the corresponding database field data types and sizes. Refer to Chapter 8 for the lists of equivalent VAX data types.

A convenient way to declare host language variables is to copy database definitions from the data dictionary, CDD/Plus, if your host language includes a data dictionary copy statement. You can copy field and relation definitions, which include all the fields within the relation. However, you must be careful to copy only those field and relation definitions with data types that are supported by your host language. See Chapter 12 and Chapter 16 for more information about using data dictionary definitions. See the documentation set for your host language to determine if your host language includes a data dictionary copy statement.

19.4.2 Using Host Language Variables

You cannot include host language variables directly in the string literal Rdb/VMS statement. Instead, you use the placeholder parameter, !VAL, to reserve a place for each of the host language variables in the string literal. (The !VAL placeholder is similar in concept to the FAO arguments that are embedded in an error message.) You then locate the corresponding host language variables in the parameter list that follows the Rdb/VMS statement string. The !VAL parameter placeholders and host language variables occur in paired sets. The first host language variable in the parameter list (after the Rdb/VMS string) replaces the first !VAL in the literal string, the second host language variable replaces the second !VAL, and so on. The call to the RDB$INTERPRET function is likely to fail if the parameter list is not *in the correct order.*

The RDB$INTERPRET function expects all parameters to be passed by descriptor. You must use the by-descriptor passing mechanism when your programming language does not by default pass a host language variable by descriptor. For example, the host language variable that receives the value of a statistical GET statement is numeric. FORTRAN passes numeric values by reference. Therefore, you must use the %DESCR passing mechanism to pass this parameter to RDB$INTERPRET.

The function RDB$INTERPRET accepts dynamic descriptors, so you may use the BASIC dynamic string data type in Callable RDO programs. In other languages, such as FORTRAN, you can call a function that defines a dynamic descriptor for a variable. Use dynamic descriptors when you do not know the length of a host language variable, such as a segmented string segment.

Example 19–2, from the sample program PLI_SAMPLE.PLI, shows a call to RDB$INTERPRET using the host language variables first_name, employee_id, and last_name.

Example 19–2 Using Host Language Variables to Retrieve a Record in Callable RDO

```
            .
            .
            .
RDB_COMMAND = 'START_STREAM ES USING E IN '
              !! 'EMPLOYEES SORTED BY E.LAST_NAME';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

FETCH_COMMAND = 'FETCH ES';
RDB_COMMAND = 'GET !VAL = E.FIRST_NAME; '
               !! '!VAL = E.EMPLOYEE_ID; '
               !! '!VAL = E.LAST_NAME END_GET;';

  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (FETCH_COMMAND));

  DO WHILE (RDB_STATUS_SUCCESS);
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                               DESCRIPTOR (first_name),
                               DESCRIPTOR (employee_id),
                               DESCRIPTOR (last_name));

    found = '0'B;

    NRDB_COMMAND = 'START_STREAM DS USING D IN '
                !! 'DEGREES WITH D.EMPLOYEE_ID = !VAL';
    NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND),
                    DESCRIPTOR (employee_id));

    NFETCH_COMMAND = 'FETCH DS';
    NRDB_COMMAND = 'GET !VAL = D.DEGREE; '
                    !! '!VAL = D.DEGREE_FIELD END_GET;';

    NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NFETCH_COMMAND));
            .
            .
            .
```

19.4.3 Using Literals

Use literal values to replace variables in the same way you would in any high-level language. Literal values can be either numeric or character strings. Remember, you must pass numeric literals by descriptor. String literals must be passed by descriptor and must be quoted with either double (" ") or single (' ') quotation marks; follow the conventions for your host language. You can use a literal in any data manipulation statement that accepts a host language variable.

Example 19–3 illustrates the use of a literal value within an RSE.

Example 19–3 Using a Literal Value Within a Record Selection Expression in Callable RDO

```
    RDB_COMMAND = 'STORE C IN CANDIDATES USING '
                  !! ' C.FIRST_NAME     = !VAL; '
                  !! ' C.LAST_NAME      = !VAL; '
                  !! ' C.MIDDLE_INITIAL = !VAL; '
                  !! ' C.CANDIDATE_STATUS   = !VAL END_STORE;';

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                  DESCRIPTOR ('Marty'),
                  DESCRIPTOR ('Silberlicht'),
                  DESCRIPTOR ('A'),
                  DESCRIPTOR ('Available in June.'));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;
```

19.4.4 Retrieving Records

When you use Callable RDO from a program to access an Rdb/VMS database, you select the records that you are interested in manipulating by gathering these records into a stream. You create this stream using the Rdb/VMS data manipulation statements. These statements use context variables to name the stream of records that you select from one or more relations.

Chapter 3 provides information on how to use the data manipulation statements to select a subset of records.

In Callable RDO, you must use the START_STREAM statement to form record streams. You cannot use the FOR statement to form record streams in Callable RDO because the FOR statement uses automatic iteration to process all the records in the record stream. However, the call to the RDB$INTERPRET function can pass only one record to or from the database. Therefore, the FOR statement will fail after the first record is processed.

Rdb/VMS provides two kinds of streams that are opened with a START_STREAM statement: a declared stream and an undeclared stream. A declared stream is one that you explicitly declare to your program with the DECLARE_STREAM statement. The DECLARE_STREAM statement includes an RSE. Therefore, the START_STREAM statement for a declared stream does not include an RSE and must be preceded by the DECLARE_STREAM statement.

The declaration of your stream with the DECLARE_STREAM statement is valid for the duration of your program. This means you cannot use the context variable that you use in the RSE of the DECLARE_STREAM statement in any other RSE within your program. However, this also means that you can declare the stream once, and then start and end it several times without having to specify the RSE again. If you use the undeclared START_STREAM statement, you must specify the RSE each time that you start the stream.

When you issue a DECLARE_STREAM statement that contains host language variables in the RSE, Rdb/VMS examines the host language variables at the time it executes the DECLARE_STREAM statement. Any changes you make to the host language variables after this statement have no effect on the records included in the stream.

An undeclared stream does not use the DECLARE_STREAM statement. Instead, you specify the RSE on the START_STREAM statement.

The undeclared START_STREAM statement always includes at least one context variable in its RSE. The scope of the context variable begins with the START_STREAM statement and ends with the END_STREAM statement. If you do not include an END_STREAM statement for a particular record stream, the scope extends to the end of the transaction. A context variable is meaningless outside its scope.

When you use an undeclared stream that contains host language variables in the RSE, Rdb/VMS examines the host language variables at the time it executes the START_STREAM statement. Any changes you make to the host language variables after this statement have no effect on the records included in the stream.

After opening a record stream created by either of the START_STREAM statements, use the FETCH statement to fetch the next record, and a GET statement to transfer the field value or values to a host language variable.

In Callable RDO, your program must explicitly detect the end-of-stream condition. When there are no more records in the stream, the FETCH statement returns the Rdb/VMS symbolic error code, RDB$_STREAM_EOF. The RDB$INTERPRET function returns this error code to your program in the return status value. Your program exits from the FETCH-GET loop and closes the stream when it detects this value. Refer to Section 19.7.4 for more information about handling the end-of-stream condition.

You can process a record stream only from the beginning. To return to a record you have already processed, you must first close the stream, open it again, and then start processing the stream again from the beginning of the stream if you are using the undeclared START_STREAM statement. If you are using a declared START_STREAM statement, you only have to issue another START_STREAM statement to return to the start of the stream and then process the stream until you reach the record you desire.

To end a declared or undeclared stream, issue the END_STREAM statement. This statement must include the same stream name used to start the stream. If you form any streams within a transaction, do not execute an END_STREAM statement after a COMMIT or ROLLBACK statement. The COMMIT and ROLLBACK statements automatically end all streams opened during that transaction. If you issue an END_STREAM statement after ending a transaction, Rdb/VMS returns the exception condition, RDO$_STRNOTOPE.

Example 19–4, from the DISPLAY_CAND function, shows the use of the START_STREAM and FETCH statements.

Example 19–4 Using the START_STREAM and FETCH Statements in Callable RDO

```
STORE_CAND: PROCEDURE;

/**************************************************************************/
/* This procedure stores a record in the CANDIDATES relation.  It shows how */
/* to store a value in a field of data type VARYING STRING.               */
/**************************************************************************/


  /* Initialize variables.*/

  continue     = 'N';
  succeed      = '1'B;
  status       = 0;
  err          = 0;
  x            = 0;
  i            = 0;
  new_count    = 0;
  candidate_status = ' ';
  status_length = 1;
  first_name   = '00000';

  /* Prompt the user for data to store in the CANDIDATES relation. */

  DO WHILE ((first_name ^= 'exit') | (first_name ^= 'EXIT'));

    DO WHILE ((continue = 'N') | (continue = 'n'));

      new_count = new_count + 1;

      PUT SKIP LIST ('Please enter the first name of the candidate'!!
                     ' or type exit');

      GET LIST (first_name);
TEST: IF first_name = 'exit' | first_name = 'EXIT' THEN
        GOTO ENDER;

      PUT SKIP LIST ('Please enter the middle initial of the candidate');
      GET LIST (middle_initial);

      PUT SKIP LIST ('Please enter the last name of the candidate');
      GET LIST (last_name);

      PUT SKIP LIST ('Please enter the candidate status information');
      GET LIST (candidate_status);
      status_length = LENGTH(TRIM(candidate_status));

      PUT SKIP LIST ('Have you entered the candidate info correctly? (Y,N) ');
      GET LIST (continue);

    END; /* while continue = n */
```

(continued on next page)

Example 19–4 (Cont.) Using the START_STREAM and FETCH Statements in Callable RDO

```
    RDB_COMMAND = 'START_TRANSACTION READ_WRITE RESERVING '
                  !! ' CANDIDATES FOR SHARED WRITE ';

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

/* Store the values specified by the user in the CANDIDATES relation. */
/* Check for errors and inform the user of the success or failure of  */
/* the STORE operation.                                               */

    RDB_COMMAND = 'STORE C IN CANDIDATES USING '
                  !! ' C.FIRST_NAME     = !VAL; '
                  !! ' C.LAST_NAME      = !VAL; '
                  !! ' C.MIDDLE_INITIAL = !VAL; '
                  !! ' C.CANDIDATE_STATUS   = !VAL END_STORE;';

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                  DESCRIPTOR (first_name),
                  DESCRIPTOR (last_name),
                  DESCRIPTOR (middle_initial),
                  DESCRIPTOR (candidate_status));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

    IF (succeed) THEN DO;
      RDB_COMMAND = 'COMMIT';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      PUT SKIP LIST ('Update operation succeeded');
      END; /* succeed */
    ELSE  DO;
      RDB_COMMAND = 'ROLLBACK';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      PUT SKIP LIST ('Update operation failed');
    END; /* else do for succeed */

    continue = 'n';

  END; /* (first_name ^= 'exit') */

ENDER: PUT SKIP;

END STORE_CAND;
```

19.4.5 Retrieving Segmented Strings

The Rdb/VMS segmented string data type allows you to store blocks of unstructured data such as text, graphics, voice, telemetry, or bit streams. You store segmented string records in a field of a relation. Each record can hold any number of segmented strings, up to the physical limits of the storage unit. Each segment can be up to 65,522 bytes long, except for the first segment of the string, which has a maximum length of 65,508 bytes. See Chapter 8 for more information on the segmented string data type.

The Rdb/VMS segmented string data type requires a the use of a pair of record selection expressions. The first RSE forms an outer stream of records. It determines the field and the relation that contain the segmented string records. The second RSE forms the inner stream of segments. It identifies the segmented string field that contains the individual segments.

In Callable RDO, you can retrieve segmented strings with the START_SEGMENTED_STRING statement. You must start two streams when processing segmented string streams with the START_SEGMENTED_STRING statement. Use the START_STREAM statement to form an outer stream of records. Then use the START_SEGMENTED_STRING statement to form an inner stream of segments. This inner stream identifies the segment stream that is contained in the field specified by the START_STREAM statement. When you name the segment stream, use a name different from the outer stream name. Use different context variables for the outer stream and the inner segmented string stream.

The inner stream is not a stream in the sense that you can control its record selection. The segmented string behaves like a sequential record file. You must begin at the first segment and retrieve segments in the order that they are stored. For this reason, the inner stream does not include selection clauses. Note that the START_SEGMENTED_STRING statement uses a segmented string variable in place of the context variable, and that the field name is qualified by the context variable specified in the outer START_STREAM statement.

Use the FETCH statement to advance the pointer in the outer START_STREAM record stream. Use the GET statement in the inner stream to retrieve the segmented string. There are two special keywords supplied by Rdb/VMS: RDB$VALUE, containing the segmented string segment just retrieved, and RDB$LENGTH, an unsigned word integer that contains the length of this segment. Within the START_SEGMENTED_STRING statement, the GET statement automatically fetches the contents of the segment, RDB$VALUE, and automatically advances the segment pointer to the next segmented string.

The GET statement fetches only as much of the stored segment as the host language variable that receives the segment can hold. If the entire segment is not retrieved by the GET statement, Rdb/VMS returns the exception condition, RDB$_SEGMENT. Your program can examine this return status value to determine if more of the segment remains. (See Section 19.7 for information on detecting and handling exception conditions in Callable RDO.) Use succeeding GET statements to fetch the remaining pieces of the segment, then fetch the second segment in the same manner, and so on. You can use dynamic descriptors to receive the segments if you do not know how big the segment is.

Example 19–5, from the DISPLAY_RESUME procedure, retrieves a segmented string. The example:

- Starts a stream RS that selects the RESUMES relation
- Fetches an EMPLOYEES record based on an EMPLOYEE_ID specified earlier in the program
- Starts a segmented string LS
- Issues a GET statement to retrieve and display the segments until there are no more segments left
- Ends the segmented string LS
- Closes the stream RS

Example 19–5 Retrieving a Segmented String with the START_STREAM and START_SEGMENTED_STRING Statements in Callable RDO

```
               .
               .
               .
RDB_COMMAND = 'START_TRANSACTION READ_ONLY RESERVING '
              !! ' RESUMES FOR SHARED READ ';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–5 (Cont.) Retrieving a Segmented String with the START_STREAM and START_SEGMENTED_STRING Statements in Callable RDO

```
/* Start a stream to retrieve the employee record   */
/* or records with the specified ID.                */

RDB_COMMAND = 'START_STREAM RS USING '
              !! 'RR IN RESUMES WITH RR.EMPLOYEE_ID = !VAL';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                           DESCRIPTOR (employee_id));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;


  RDB_COMMAND = 'FETCH RS';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  /* Use a START_SEGMENTED_STRING statement to retrieve    */
  /* the individual segments that comprise the segmented  */
  /* string.                                               */


  NRDB_COMMAND = 'START_SEGMENTED_STRING LS USING L IN RR.RESUME ';
  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND));
  IF (^NRDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;


  NRDB_COMMAND = 'GET !VAL = L.RDB$VALUE; END_GET; ';
  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND),
                                    DESCRIPTOR (resume_segment));
  DO WHILE (NRDB_STATUS = 1);

    PUT SKIP LIST (TRIM(resume_segment));
    resume_segment = ' ';
    NRDB_COMMAND = 'GET !VAL = L.RDB$VALUE END_GET; ';
    NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND),
                                 DESCRIPTOR (resume_segment));

  END;  /* NRDB_STATUS = 1  */
  NRDB_COMMAND = 'END_SEGMENTED_STRING LS';
  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND));
  IF (^NRDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

RDB_COMMAND = 'END_STREAM RS';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–5 (Cont.) Retrieving a Segmented String with the START_STREAM and START_SEGMENTED_STRING Statements in Callable RDO

```
  RDB_COMMAND = 'COMMIT';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

   continue = 'n';

 END; /* (employee_id ^= 'exit') */

ENDER: PUT SKIP;

END DISPLAY_RESUME;
```

19.4.6 Retrieving Field and Statistical Values

Use the GET statement to retrieve one, several, or all the fields in a database record. You can also use the GET statement to retrieve statistical and Boolean values from the database.

The GET statement is a read operation; you may include it in any Rdb/VMS transaction. However, you should include the GET statement in a read/write transaction if you require a highly accurate picture of the database, or if you intend to update the database using the values returned by the GET statement.

Section 19.4.6.1 and Section 19.4.6.2 discuss retrieving field and statistical values.

19.4.6.1 Using the GET Statement to Retrieve Field Values When you form a record stream using the START_STREAM statement, you include the FETCH and GET statements between the START_STREAM and END_STREAM statements. An earlier example, Example 19–2, shows the use of the GET statement to retrieve field values.

19.4.6.2 Using the GET Statement to Retrieve Statistical Values You can retrieve the result of a statistical expression directly without processing each record in the record stream. The result of a statistical expression is an aggregate and is often not the same data type as the field to which it refers.

Note *The result returned by a GET statement when it is used with statistical expressions is always numeric. The RDB$INTERPRET function passes all parameters by descriptor. However, most programming languages do not pass numeric values by descriptor. Include a by-descriptor passing mechanism for any field that receives the result of a statistical GET statement.*

Example 19–6, from the STATS procedure, shows a PL/I call to RDB$INTERPRET with the COUNT statistical expression.

Example 19–6 Using the GET Statement to Retrieve a Statistical Value in Callable RDO

```
STATS: PROCEDURE;

/* This procedure displays the total number of records stored in */
/* the EMPLOYEES relation.                                       */

  /* Initialize variables. */

  err          = 0;

  RDB_COMMAND = 'START_TRANSACTION READ_ONLY ';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

  /* Use the GET statement with a statistical function to   */
  /* calculate the total number of records in the EMPLOYEES */
  /* relation.                                              */

  RDB_COMMAND = 'GET !VAL = COUNT OF E IN EMPLOYEES END_GET;';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (I));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

  /* Display the value. */

  PUT SKIP LIST ('The number of employees in the Corporation is: ', I);

  RDB_COMMAND = 'ROLLBACK ';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

END STATS;
```

19.4.7 Updating Records

RDO update statements can only be used within a read/write transaction. (You may, of course, include any valid RDO statement within a read/write transaction.) The update statements that require a read/write transaction are:

- STORE
- MODIFY
- ERASE

Also, include the GET statement in a read/write transaction if you intend to update any of the fields returned by the GET statement.

Note *You may not use a view to update records if that view refers to more than one relation. Furthermore, do not update a field that is mentioned after the keyword OVER in a CROSS clause. You should consider fields over which relations are joined (crossed) as available for read-only access.*

If your program prompts for data from a terminal, consider nesting a complete transaction within the data-input loop. If the transaction fails, the user needs to enter only the last input data again. But keep in mind that the transaction should be short and, at the same time, logically must not only partially update the database.

19.4.7.1 Storing Records You can use a single STORE statement to insert values into one or more fields in one relation. To store more than one record in a relation, include the STORE statement within a program loop. Example 19–7, from the ADD_EMPLOYEES procedure, stores a new EMPLOYEES record.

Example 19–7 Storing Records in Callable RDO

```
ADD_EMPLOYEES: PROCEDURE;

/*****************************************************************/
/* This procedure adds a new EMPLOYEES record to the EMPLOYEES   */
/* relation.                                                     */
/*****************************************************************/

  /* Initialize variables. */

  employee_id  = '00000';
  continue     = 'N';
  succeed      = '1'B;
  ascii_bday   = '        ';
  status       = 0;
  err          = 0;
  x            = 0;
  see_all      = 'N';
  DO i = 1 TO 5;
    db_key_array(i) = '        ';
  END; /* do 1 = 1 to 5 */
  i            = 0;
  new_count    = 0;


/* Prompt user for input, until user enters 'exit'. */

  DO WHILE ((employee_id ^= 'exit') | (employee_id ^= 'EXIT'));

    DO WHILE ((continue = 'N') | (continue = 'n'));

      new_count = new_count + 1;

      PUT SKIP LIST ('Please enter the ID of the new Employee or type exit');
      GET LIST (employee_id);
```

(continued on next page)

Example 19–7 (Cont.) Storing Records in Callable RDO

```
TEST: IF employee_id = 'exit' | employee_id = 'EXIT' THEN
        GOTO ENDER;

      PUT SKIP LIST ('Please enter the Employees last name');
      GET LIST (last_name);

      PUT SKIP LIST ('Please enter the Employees first name');
      GET LIST (first_name);

      PUT SKIP LIST ('Please enter the Employees middle initial');
      GET LIST (middle_initial);

      /* Prompt user to enter date, keep prompting until user */
      /* enters the date in the proper format.                */

      PUT SKIP LIST ('Please enter the Employees birthday');
      PUT SKIP LIST ('In this format: 14-AUG-1956 0:0:0.0');
      GET LIST (ascii_bday);

      PUT SKIP LIST ('Please enter the Employees sex');
      GET LIST (sex);

      PUT SKIP LIST ('Please enter the Employees street address');
      GET LIST (address_data_1);

      PUT SKIP LIST ('Please enter the Employees apartment number, if any');
      GET LIST (address_data_2);

      PUT SKIP LIST ('Please enter city');
      GET LIST (city);

      PUT SKIP LIST ('Please enter state');
      GET LIST (state);

      PUT SKIP LIST ('Please enter postal code');
      GET LIST (postal_code);

      PUT SKIP LIST ('Please enter status code');
      GET LIST (status_code);

      PUT SKIP LIST ('Have you entered all data correctly? (Y,N) ');
      GET LIST (continue);

    END; /* while continue = n */


    /* Pass the START_TRANSACTION statement to RDB$INTERPRET. */

    RDB_COMMAND = 'START_TRANSACTION READ_WRITE RESERVING '
                  !! ' EMPLOYEES FOR SHARED WRITE ';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

    PUT SKIP LIST ('Progressing to STORE');
```

(continued on next page)

Example 19–7 (Cont.) Storing Records in Callable RDO

```
/* Store the values in the EMPLOYEES relation. */

RDB_COMMAND = 'STORE E IN EMPLOYEES USING '
              !! ' E.EMPLOYEE_ID    = !VAL; '
              !! ' E.LAST_NAME      = !VAL; '
              !! ' E.FIRST_NAME     = !VAL; '
              !! ' E.MIDDLE_INITIAL = !VAL; '
              !! ' E.BIRTHDAY       = !VAL; '
              !! ' E.SEX            = !VAL; '
              !! ' E.ADDRESS_DATA_1 = !VAL; '
              !! ' E.ADDRESS_DATA_2 = !VAL; '
              !! ' E.CITY           = !VAL; '
              !! ' E.STATE          = !VAL; '
              !! ' E.POSTAL_CODE    = !VAL; '
              !! ' E.STATUS_CODE    = !VAL END_STORE;';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
              DESCRIPTOR (employee_id),
              DESCRIPTOR (last_name),
              DESCRIPTOR (first_name),
              DESCRIPTOR (middle_initial),
              /* RDB$INTERPRET does DATE conversion. */
              DESCRIPTOR (ascii_bday),
              DESCRIPTOR (sex),
              DESCRIPTOR (address_data_1),
              DESCRIPTOR (address_data_2),
              DESCRIPTOR (city),
              DESCRIPTOR (state),
              DESCRIPTOR (postal_code),
              DESCRIPTOR (status_code));

IF (^RDB_STATUS_SUCCESS) THEN
    CALL HANDLE_ERROR;

IF succeed THEN DO;
  RDB_COMMAND = 'START_STREAM ES USING '
               !! 'E IN EMPLOYEES WITH E.EMPLOYEE_ID = !VAL';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (employee_id));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  RDB_COMMAND = 'FETCH ES';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
     CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–7 (Cont.) Storing Records in Callable RDO

```
  /* Get the dbkey associated with the newly stored EMPLOYEES record. */

  i = i + 1;
  RDB_COMMAND = 'GET !VAL = E.RDB$DB_KEY END_GET';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (db_key_array(i)));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

RDB_COMMAND = 'END_STREAM ES';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
IF (^RDB_STATUS_SUCCESS) THEN
    CALL HANDLE_ERROR;


/* If the user wants to see all of the EMPLOYEES records   */
/* added during this session, step through the array of    */
/* dbkeys to find and print each new EMPLOYEES record.     */

PUT SKIP LIST ('Successfully added employee: ',last_name);
PUT SKIP LIST (' with employee id: ',employee_id);
PUT SKIP LIST ('Do you want to see the names of all the');
PUT SKIP LIST ('employees entered during this session? (Y,N)');
GET LIST (see_all);

IF see_all = 'Y' | see_all = 'y' THEN DO;
  DO x = 1 TO i;
  RDB_COMMAND = 'START_STREAM ED USING '
              !! ' E IN EMPLOYEES WITH E.RDB$DB_KEY = !VAL';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (db_key_array(x)));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;


  RDB_COMMAND = 'FETCH ED';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  RDB_COMMAND = 'GET !VAL = E.FIRST_NAME; '
               !! '!VAL = E.LAST_NAME END_GET';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (first_name),
                             DESCRIPTOR (last_name));

  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  PUT SKIP LIST (first_name, ' ', last_name);

  RDB_COMMAND = 'END_STREAM ED';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  END; /* for x */

END; /* see_all */
```

(continued on next page)

Example 19–7 (Cont.) Storing Records in Callable RDO

```
      RDB_COMMAND = 'COMMIT';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      END; /* succeed */
    ELSE  DO;
      PUT SKIP LIST ('Update operation failed, ',last_name);
      PUT SKIP LIST (' with employee ID: ',employee_id);
      PUT SKIP LIST (' has not been stored in the database');

      RDB_COMMAND = 'ROLLBACK';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

    END; /* else do for not success */

    continue = 'n';

  END; /* (employee_id ^= 'exit') */

ENDER: PUT SKIP;

END ADD_EMPLOYEES;
```

Use the CREATE_SEGMENTED_STRING statement and the STORE statement to store segmented strings in a relation. You must use two operations when storing segmented strings. The order in which you process the inner and outer STORE operations is the reverse order of segmented string retrieval.

First, use the CREATE_SEGMENTED_STRING statement to form the inner string of segments. Store the segments in this inner string with the STORE statement. Your program must explicitly repeat the STORE statement to store each segment, or iterate the STORE statement by a program loop. You cannot selectively store individual segments, and you must store the segmented string in its entirety. For example, if you attempted to store first segment-1, next segment-3, next segment-5, and finally segment-2, your segmented string would contain: segment-1, segment-3, segment-5, and segment-2, in that order.

When all the segments are stored into a segmented string, use an outer STORE statement to store the segmented string identifier into a relation. (The value you store in the relation is in fact a pointer to the segmented string.) You can store other fields in the relation with the same STORE statement. Once the outer store operation is complete, close the segmented string with the END_SEGMENTED_STRING statement.

You can close the segmented string before you perform the outer store operation in order to store the segmented string identifier in a relation. However, do not use that segmented string identifier again until you have stored it in a relation.

Example 19–8, from the STORE_RES procedure, stores a segmented string. This example:

- Prompts the user for input
- Creates a segmented string RS_HANDLE
- Uses a STORE statement to store each segment S.RDB$VALUE into RS_HANDLE
- Uses another STORE statement to store the fields RS_HANDLE (the pointer to the segmented string) and employee_id in RESUMES
- Uses an END_SEGMENTED_STRING statement to close the segmented string RS_HANDLE

Example 19–8 Using the CREATE_SEGMENTED_STRING Statement in Callable RDO

```
STORE_RES: PROCEDURE;

/*********************************************************/
/* This procedure demonstrates how to store a record with */
/* a field of data type SEGMENTED STRING.                */
/*********************************************************/

  DECLARE  resume_segment1 CHARACTER(80) VARYING;
  DECLARE  mfile_name     CHARACTER(10) VARYING;
  DECLARE  my_file        FILE;
  DECLARE  end_of_file    BIT;

  /* Initialize variables. */

  employee_id  = '00000';
  continue     = 'N';
  correct      = 'N';
  succeed      = '1'B;
  resume_segment1 = ' ';
  end_of_file  = '0'B;
  status       = 0;
  err          = 0;
  i            = 0;

  /* Prompt the user for the employee ID of the RESUMES record */
  /* he or she wants to modify.                                */
```

(continued on next page)

Example 19–8 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in Callable RDO

```
  DO WHILE ((employee_id ^= 'exit') | (employee_id ^= 'EXIT'));

    DO WHILE ((continue = 'N') | (continue = 'n'));

      new_count = new_count + 1;

      PUT SKIP LIST ('Please enter the ID of the new Employee'!!
                     'or type exit');

      GET LIST (employee_id);

TEST: IF employee_id = 'exit' | employee_id = 'EXIT' THEN
        GOTO ENDER;

      PUT SKIP LIST ('Please enter file name of new resume');
      GET LIST (mfile_name);

      PUT SKIP;
      PUT SKIP LIST ('Have you entered all data correctly? (Y,N) ');
      GET LIST (continue);

    END; /* while continue = n */

    OPEN FILE(my_file) TITLE(mfile_name) SEQUENTIAL;
    ON ENDFILE(my_file) end_of_file = '1'B;

    RDB_COMMAND = 'START_TRANSACTION READ_WRITE RESERVING '
                  !! ' RESUMES FOR SHARED WRITE ';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;


    RDB_COMMAND = 'CREATE_SEGMENTED_STRING RS_HANDLE;';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;


    READ FILE(my_file) INTO (resume_segment1);
    DO WHILE (^end_of_file);
      RDB_COMMAND = 'STORE L IN RS_HANDLE USING '
                    !! ' L.RDB$VALUE = !VAL END_STORE;';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                    DESCRIPTOR (TRIM(resume_segment1)));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      READ FILE(my_file) INTO (resume_segment1);
    END; /* do while not eof */
```

(continued on next page)

Example 19–8 (Cont.) Using the CREATE_SEGMENTED_STRING Statement in Callable RDO

```
    RDB_COMMAND = 'STORE RE IN RESUMES USING '
                  !! ' RE.RESUME         = RS_HANDLE;'
                  !! ' RE.EMPLOYEE_ID    = !VAL END_STORE;';

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                  DESCRIPTOR (employee_id));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

    RDB_COMMAND = 'END_SEGMENTED_STRING RS_HANDLE;';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));

    IF succeed THEN DO;
      RDB_COMMAND = 'COMMIT';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      PUT SKIP LIST ('Update operation succeeded');
    END; /* succeed */
  ELSE DO;
      RDB_COMMAND = 'ROLLBACK';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

      PUT SKIP LIST ('Update operation failed');
    END; /* else not succeed */

    continue = 'n';

    CLOSE FILE(my_file);

  END; /* (employee_id ^= 'exit') */

ENDER: PUT SKIP;

END STORE_RES;
```

19.4.7.2 Modifying Records

Using a single MODIFY statement, you can change values in one or more fields of a record in a relation. When you list fields in the MODIFY statement, list only those fields that you want to change. If you replace a field value with an identical field value, you are needlessly adding overhead to your program.

Before modifying records, you must start a read/write transaction and form a record stream that contains the records you wish to modify.

Use the START_STREAM statement when you want to modify the records in the record stream. The START_STREAM statement allows you to conditionally modify a record that has been fetched. Use host language variables within your RSE so your program logic can alter a record stream for each new START_STREAM statement.

Example 19–9, from the MODIFY_EMPLOYEES procedure:

- Prompts the user for input
- Starts a stream ES that contains the EMPLOYEES record for the employee whose ID is the same value as the host variable, employee_id
- Fetches the record
- Changes the current address to the values entered by the user
- Closes the stream ES

Example 19–9 Modifying Records in Callable RDO

```
                .
                .
                .
      PUT SKIP LIST ('Please enter new street address');
      GET LIST (address_data_1);

      PUT SKIP LIST ('Please enter new box number or apt number');
      GET LIST (address_data_2);

      PUT SKIP LIST ('Please enter the city');
      GET LIST (city);

      PUT SKIP LIST ('Please enter the state');
      GET LIST (state);

      PUT SKIP LIST ('Please enter the postal code');
      GET LIST (postal_code);

      PUT SKIP LIST ('Have you entered the address correctly? (Y,N) ');
      GET LIST (correct);
                 .
                 .
                 .
    RDB_COMMAND = 'START_TRANSACTION READ_WRITE RESERVING '
                  !! ' EMPLOYEES FOR SHARED WRITE ';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–9 (Cont.) Modifying Records in Callable RDO

```
  /* Modify the address fields for the specified employee. */

  RDB_COMMAND = 'START_STREAM ES USING '
              !! 'E IN EMPLOYEES WITH E.EMPLOYEE_ID = !VAL';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (employee_id));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

  RDB_COMMAND = 'FETCH ES';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;


    RDB_COMMAND = 'MODIFY E USING'
                !! ' E.ADDRESS_DATA_1 = !VAL;'
                !! ' E.ADDRESS_DATA_2 = !VAL;'
                !! ' E.CITY = !VAL;'
                !! ' E.STATE = !VAL;'
                !! ' E.POSTAL_CODE = !VAL END_MODIFY;';

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (address_data_1),
                             DESCRIPTOR (address_data_2),
                             DESCRIPTOR (city),
                             DESCRIPTOR (state),
                             DESCRIPTOR (postal_code));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

  RDB_COMMAND = 'END_STREAM ES';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

  /* Notify user of the success or failure of the MODIFY operation. */

  IF succeed THEN DO;
    RDB_COMMAND = 'COMMIT';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;

    PUT SKIP LIST ('Update operation succeeded');
  END; /* succeed */
ELSE DO;
    RDB_COMMAND = 'ROLLBACK';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
    IF (^RDB_STATUS_SUCCESS) THEN
        CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–9 (Cont.) Modifying Records in Callable RDO

```
SKIP_MOD:
      PUT SKIP LIST ('Update operation failed');
    END; /* else not succeed */
              .
              .
              .
```

19.4.7.3 Modifying Segmented Strings To modify a segmented string you must first create a new segmented string with the CREATE_SEGMENTED_STRING statement and then modify the existing record by replacing the logical pointer to the old segmented string identifier with the logical pointer to the new segmented string identifier. As Chapter 8 explains in more detail, when you store a segmented string field, you do not actually store segments into a record; you store a logical pointer to the first segment in the segmented string. Thus, by creating a new segmented string and a new segmented string identifier associated with it, you can modify the field in a database record that "contains" a segmented string merely by replacing the old segmented string identifier with a new segmented string identifier.

Note *Although you use a MODIFY statement to modify segmented strings, you are not actually modifying the individual segments that comprise the segmented string field. You are actually replacing the entire segmented string field value with a new segmented string value. Example 19–8 demonstrates how this is done.*

19.4.7.4 Erasing Records You can delete one, many, or all the records from a relation using the ERASE operation. Before erasing records, you must start a read/write transaction and form a record stream that contains the records you wish to erase.

The ERASE statement can be an extremely expensive operation, using almost as many system resources as a load operation. In shared and protected share modes, each record erased generates a record in both the recovery-unit journal and the after-image journal. Thus, large-scale erasing of database records may exceed the enqueue limit (ENQLM). See the *VAX Rdb/VMS Guide to Database Maintenance and Performance* for information on modifying system resources.

Use the START_STREAM statement when you want to erase multiple records in a relation. The START_STREAM statement lets you conditionally erase the record that has been fetched. You can display values from selected fields in this record and then decide to erase it or not.

Use host language variables within your RSE so your program logic can alter a record stream for each new START_STREAM statement.

Example 19–10, from the DELETE_RECORD procedure, erases a record from the JOB_HISTORY relation.

Example 19–10 Erasing Records in Callable RDO

```
RDB_COMMAND = 'START_STREAM JS USING '
              !! 'JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = !VAL';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                           DESCRIPTOR (employee_id));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

DO WHILE (succeed);
RDB_COMMAND = 'FETCH JS';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

IF Succeed THEN DO;
RDB_COMMAND = 'ERASE JH;';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
       CALL HANDLE_ERROR;
END;
END; /* end while succeed */

succeed      = '1'B;

RDB_COMMAND = 'END_STREAM JS';
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
  IF (^RDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;
```

19.5 Using Rdb/VMS Data Definition Statements

Depending on your particular application, you may need to perform some or all of the following data definition functions:

- Define fields, relations, views, indexes, constraints, or triggers
- Change fields, relations, storage areas
- Delete fields, relations, views, indexes, constraints, triggers, or storage areas

Data definition statements in Callable RDO are identical to data definition statements in interactive RDO. You can include any data definition statements from an RDO prototype in your Callable RDO program without change. Simply pass these data definition statements as literal strings to the RDB$INTERPRET function. Note that unsuccessful data definition calls return a different set of symbolic error codes than unsuccessful data manipulation calls.

If the first executable statement in your program is a data definition statement, Rdb/VMS starts a read/write transaction. You can include any valid Rdb/VMS statement within this transaction, but be aware that a lengthy read/write transaction can limit other users' access to the database.

When you design your data definition transactions, consider the effect of an unexpected program termination. If your data definition is done in discrete transactions, an early program termination may leave the database with altered metadata. You can include a call to delete this metadata in your program's cleanup routines.

Example 19–11, from the DDL_STMNT procedure, defines a temporary index for the EMPLOYEES relation. Note that you should not perform this type of operation while others are using the database.

Example 19–11   Using Data Definition Statements in Callable RDO

```
DDL_STMNT: PROCEDURE;

/*******************************************************************/
/* This procedure demonstrates how to perform data definition tasks. */
/*******************************************************************/
DECLARE literal   CHARACTER(100);

  /* Initialize variables. */

  employee_id  = ’00000’;
  continue     = ’N’;
  correct      = ’N’;
  succeed      = ’1’B;
  status       = 0;
  err          = 0;
  i            = 0;

  DO WHILE ((literal ^= ’exit’) | (literal ^= ’EXIT’));

    DO WHILE ((continue = ’N’) | (continue = ’n’));

     /* Prompt user for input.  Ordinarily, it would not be likely that */
     /* you would ask a user to define an index for the database.      */
     /* This example serves only to show you how this type of task can  */
     /* be done within a program environment.                           */
```

(continued on next page)

```
      PUT SKIP;
      PUT SKIP;
      PUT SKIP LIST ('Please enter the data definition statement to define');
      PUT SKIP LIST (' or delete a temporary index, or type exit');
      PUT SKIP LIST ('For example, to define an index for EMPLOYEES based');
      PUT SKIP LIST (' on EMPLOYEE_ID, you might enter: ');
      PUT SKIP LIST ('define index emp_employee_id for');
      PUT SKIP LIST ('employees.employee_id.');
      PUT SKIP LIST (' end index. NOTE: ENCLOSE IN SINGLE QUOTES');
      PUT SKIP LIST ('To delete this index, you might enter: ');
      PUT SKIP LIST (' delete index emp_employee_id.');
      PUT SKIP;
      GET LIST (literal);
TEST: IF literal = 'exit' | literal = 'EXIT' THEN
        GOTO ENDER;

      PUT SKIP;
      PUT SKIP LIST ('Did you enter the definition correctly (Y,N)');
      GET LIST (continue);
      PUT SKIP;

    END; /* while continue = n */


    /* Start a READ_WRITE transaction. */

    RDB_COMMAND = 'START_TRANSACTION READ_WRITE; ';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

    RDB_COMMAND = literal;

    /* Pass the data definition statement specified by the user to  */
    /* RDB$INTERPRET.                                                */

    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;


    /* Inform the user of the success or failure of the data definition */
    /* task.                                                            */

    IF succeed THEN DO;
      RDB_COMMAND = 'COMMIT';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;


      PUT SKIP LIST ('Operation succeeded');
    END; /* succeed */
  ELSE DO;
      RDB_COMMAND = 'ROLLBACK';
      RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;
```

(continued on next page)

Example 19–11 (Cont.) Using Data Definition Statements in Callable RDO

```
      PUT SKIP LIST ('Operation failed');
    END; /* else not succeed */

    continue = 'n';

  END; /* (literal ^= 'exit') */

ENDER: PUT SKIP;

END DDL_STMNT;
```

19.6 Mixing Preprocessed and Callable RDO Statements in a Single Transaction

If you want or need to include preprocessed statements and Callable RDO statements in the same transaction, the recommended technique is:

1. Use the DATABASE statement with a database handle in the preprocessed module to declare information to the preprocessor. This is not an executable statement.
2. Use the DATABASE statement in the call to RDB$INTERPRET to establish the database context and to attach to the database. This is an executable statement and should be the first statement to access the database. Use the !VAL parameter and have the database handle returned in the variable specified (declared) in Step 1.

   You should not use a preprocessed READY statement, as this will attempt to attach to an already attached (open) database.
3. Start a transaction in the preprocessed module, if possible. This is more efficient than calling RDB$INTERPRET, which must parse the command string.

   Be sure to save the transaction handle because it is used to keep the Callable RDO and preprocessed DATABASE statements operating in the same transaction and hence on the same database attach.

   This transaction handle should be used in all transactions, both preprocessed and Callable RDO. This includes the COMMIT and ROLLBACK statements.

Example 19–12 is a simple RDML/Pascal application that demonstrates this technique.

Example 19–12 Using Preprocessed and Callable RDO Statements in a Single Transaction

```
program COEXIST(output);

   {
   | This program uses the RDB$INTERPRET function and precompiled
   | DML statements that share the database and transaction handles.
   | That is, only one database attach is required for the application.
   |
   | This shipping database is a small one:
   |
   |           define database shipping.
   |           define field port_num datatype signed longword.
   |           define relation port.
   |               port_num.
   |           end.
   |           store p in port using p.port_num=1 end_store
   |           commit
   }

database
    db1 = filename 'SHIPPING';

function RDB$INTERPRET
        {-------------}
         (command: [class_s,readonly] packed array [l1..u1:integer] of char;
          %IMMED argument: [list] integer
         ): integer; external;

procedure RDB$SIGNAL; external;
         {----------}

procedure LIB$SPAWN
         {---------}
          (command: [class_s] packed array [l1..u1:integer] of char
                    := %IMMED 0
          ); external;

var
    t1: RDML$HANDLE_TYPE := NIL;
    sts: integer := 0;

begin
    { Attach to the database using the RDB$INTERPRET function.}
    sts := RDB$INTERPRET('database !VAL = filename "SHIPPING"',
                         %DESCR db1);
    if not ODD(sts) then RDB$SIGNAL;

    { Start a transaction in RDML and establish the transaction handle. }
    start_transaction (transaction_handle t1) read_write;

    writeln('First Pass...');
    for (transaction_handle t1) p in port
        writeln(p.port_num);
    end_for;
```

(continued on next page)

Example 19–12 (Cont.) Using Preprocessed and Callable RDO Statements in a Single Transaction

```
    {
    | Use the transaction handle to pass transaction and attach
    | information to the RDB$INTERPRET function.  This interpreted
    | string can be an arbitrary DML command string.
    }
    sts := RDB$INTERPRET('store (transaction_handle !VAL) ' +
                         'p in port using p.port_num = 25 end_store',
                         %DESCR t1);
    if not ODD(sts) then RDB$SIGNAL;

    writeln('Second Pass...');
    for (transaction_handle t1) p in port
        writeln(p.port_num);
    end_for;

    {
    | Spawn a process and use RMU to display the number of active
    | users.  There should be only one with the process ID of the
    | current user.
    }
    writeln('Spawn sub-process to examine database attaches');
    LIB$SPAWN('RMU/DUMP/USERS SHIPPING');

    rollback (transaction_handle t1);

    finish db1;

end.
```

19.7 Handling Rdb/VMS Errors

The Callable RDO program interface lets you access an Rdb/VMS database with any VAX program language that conforms to the VAX Procedure Calling Standard. Your program calls the RDB$INTERPRET function with RDO statements and host language variables as parameters; RDB$INTERPRET passes these statements to RDO, the interactive facility of Rdb/VMS.

Error handling in Callable RDO programs and in preprocessed programs differs in the following ways:

- Detecting errors

  Errors are detected by checking the return status value of each call to RDB$INTERPRET.

- Evaluating the symbolic error codes

  Some Callable RDO symbolic error codes are different from preprocessed Rdb/VMS symbolic error codes.

- Displaying error messages

  Callable RDO programs call RDB$SIGNAL, not LIB$SIGNAL.

- Error recovery

  Callable RDO programs must detect and handle the normal end-of-stream condition.

The major difference between error handling in Callable RDO programs and preprocessed programs is that preprocessed programs use the Rdb/VMS message vector, whereas Callable RDO programs do not. Rdb/VMS returns error conditions in the form of a 20-longword signal argument vector. The second longword of the signal argument vector contains the primary return status condition value, which indicates if the call succeeded and, if not, the error condition. The remaining longwords contain FAO arguments included in the error message and any secondary error messages.

In preprocessed programs, the preprocessor declares the message vector, RDB$MESSAGE_VECTOR, in the host language program. When an exception or error condition occurs, Rdb/VMS passes the return status condition value and any FAO arguments to the host language program in RDB$MESSAGE_VECTOR. Thus, the message vector is available to preprocessed programs for error handling.

In Callable RDO programs, the Rdb/VMS message vector is not passed through the RDB$INTERPRET function, and thus is not accessible to the calling program. Instead, RDB$INTERPRET returns the return status condition value of the call to RDO. For this reason, Callable RDO programs usually use the SYS$GETMSG system service rather than the SYS$PUTMSG system service, and the RDB$SIGNAL routine rather than the LIB$SIGNAL routine.

This section describes:

- Error detection by testing the return status value of the call to the RDB$INTERPRET function
- Determining which error has occurred using the LIB$MATCH_COND routine
- Error message display using the SYS$GETMSG system service and the RDB$SIGNAL routine
- Error recovery, the fatal error case, and continuing despite a fatal error

See Chapter 10 for information on how to display your own error messages.

19.7.1 Detecting Errors in Callable RDO Programs

A call to the RDB$INTERPRET function returns a condition value. If the call is a success, Rdb/VMS returns SS$_SUCCESS as the condition value. If the call fails, Rdb/VMS returns the systemwide symbolic error code that identifies the error or exception condition. The success or failure of a call is determined by the low-order bit of the return status condition value. If the call succeeds, the low-order (zero) bit of the return status value is set to 1. If the call fails, the low-order bit is not set and equals 0. Your program tests for errors by determining if the low-order bit of the return status value is set.

If the low-order bit of the return status value is not equal to 1, your program must handle the error, either by inline code or by branching to an error handler. You can determine the value of the low-order bit by using the logical operator AND and the conditional IF statement.

In PL/I, if RDB_STATUS_SUCCESS is declared as the following:

```
DECLARE RDB_STATUS         FIXED BINARY(31), /* status value */
        1 RDB_STATUS_FIELDS BASED (ADDR(RDB_STATUS)),
          2 RDB_STATUS_SUCCESS BIT(1),       /* low-order bit */
          2 RDB_STATUS_REST    BIT(31);      /* bits 1 through 32 */
```

The test would look like this:

```
IF (^RDB_STATUS_SUCCESS) THEN
    CALL HANDLE_ERROR;
```

In BASIC, the test would look like this:

```
IF (Return_stat AND 1%) = 0%
   THEN
      GO TO ERROR_HANDLER
END IF
```

In Pascal, the test would look like this:

```
if not odd(status) then
   error_handler(status);
```

In C, the test would look like this:

```
if (status & 1) == 0)
    error_handler(status);
```

In FORTRAN, the test would look like this:

```
IF ((status .AND. 1) .EQ. 0) THEN
    CALL error_handler(status)
```

Some programming languages support other conditional tests, such as SUCCESS/FAILURE. In COBOL, success/fail error detection looks like this:

```
SET RETURN-STAT TO SUCCESS.
```

```
(call to RDB$INTERPRET)
```

```
IF RETURN-STAT EQUAL FAILURE
   THEN
      PERFORM ERROR-HANDLER
END-IF.
```

The three low-order bits in the return status value together give the severity level of the error. Table 19–1 illustrates the setting of the low-order bits for each severity level.

Table 19–1 Severity Levels of the Return Status Value

Low-Order Bits			Severity	Type of
2	**1**	**0**	**Level**	**Return Status**
0	0	0	0	WARNING
0	0	1	1	SUCCESS
0	1	0	2	ERROR
0	1	1	3	INFORMATIONAL
1	0	0	4	SEVERE/FATAL

Informational error messages return a status value whose low-order bit is set to 1, indicating that the call executed successfully. However, the return status longword value is not equal to 1 because additional bits have been set. For this reason, it is unwise to determine the success of the call by the simple equality test:

```
IF Return_stat <> 1%       !BAD CODE
```

Instead, test for both the success and informational return status values, or use your programming language's conditional success/failure test.

The following example shows the program logic you should use to detect for errors in Callable RDO:

```
IF STATUS-RESULT IS FAILURE
THEN
    PERFORM LOCK-ERROR-CHECK
    GO TO GETCHK-EMP-EXIT
END-IF
```

19.7.2 Determining Which Errors Have Occurred

Errors that occur when Callable RDO statements are executed may be caused by any of the following:

- Incorrect RDO syntax
- Incorrect parameters
- Incorrect parameter passing mechanisms
- Actual run-time errors

It is easy to confuse actual run-time errors with program bugs in Callable RDO. You should use the interactive Rdb/VMS facility RDO to test your RDO statements before including them in your Callable RDO program. Then debug your program thoroughly to eliminate any incorrect parameters or parameter passing mechanisms.

After your program is fully debugged, if you detect that a call to RDB$INTERPRET has failed, you should determine which error has occurred. Your program error handler can then take the correct action for recovery or orderly program termination. You determine which error has occurred by evaluating the symbolic code associated with the error.

19.7.2.1 Using Symbolic Error Codes Every unsuccessful call to the RDB$INTERPRET function returns a status value that identifies a specific error condition. An Rdb/VMS symbolic error code is associated with each unique return status value. For example, RDB$_DEADLOCK is the symbolic error code that indicates a transaction is deadlocked.

The RDB$INTERPRET function can return two types of symbolic error codes. Each type is identified by a different prefix that has a different facility code associated with it:

- RDB$_

  The data manipulation error codes. These are the same as those codes returned by Rdb/VMS preprocessed programs. However, there are a few exceptions. For example, in Callable RDO the symbolic error code for segmented string end-of-file condition (EOF) is RDO$_NO_MORSEG, not the preprocessed error code RDB$_SEGSTR_EOF.

- RDO$_

  The data definition error codes. These codes are the same as those returned by the interactive RDO facility.

See Table A–1 in Appendix A for a list of commonly used Rdb/VMS symbolic error codes for data manipulation statements. The symbolic error codes that are unique to Callable RDO programs are marked in the table. See Table A–2 in Appendix A for a list of commonly used Rdb/VMS symbolic error codes for data definition statements. Table A–1 and Table A–2 are not exhaustive lists; you might want to create a list of likely and less likely errors for your particular type of application or programming facility. (The *VAX Rdb/VMS RDO and RMU Reference Manual* contains pointers to the online Rdb/VMS error message explanation files.)

In Callable RDO programs, you name and declare the variable that receives the return status value. You can use the symbolic error codes to control program logic for specific errors. When your program detects an error, your error handler:

- Evaluates the symbolic error code by:
  - Calling the LIB$MATCH_COND routine
  - Using a local host language equality test
- Then directs program logic with a host language multipath statement; for example, the Pascal CASE statement.

Although these symbolic names, such as RDB$_DEADLOCK, represent actual values, you should use only the symbolic names in your programs. This is because:

- The symbolic error codes themselves are mnemonic. You can assign your own mnemonic names in some programming languages.
- The VMS Linker determines the numeric values.
- If the numeric value of a symbolic error code ever changes, all you have to do is link your program again; on the other hand, if you use hardcoded values, you have to search for and change every occurrence of the value.

19.7.2.2 Declaring Symbolic Error Codes Rdb/VMS error code values are longwords and may be declared either as variables or constants. The exact format for declaring symbolic error codes is language-specific. A declaration in PL/I is shown in the next example.

```
DECLARE (RDB$_STREAM_EOF,
         RDB$_NO_RECORD,
         RDB$_DEADLOCK,
         RDB$_BAD_SEGSTR_HANDLE,
         RDB$_LOCK_CONFLICT,
         RDB$_INTEG_FAIL,
         RDB$_NO_DUP,
         RDB$_NOT_VALID,
         RDO$_DATCONERR,
         RDO$_INDNOTDEF) GLOBALREF FIXED BINARY (31);
```

Refer to the language-specific chapters for the declaration in BASIC, C, COBOL, FORTRAN, or Pascal.

Refer to your programming language user's guide if you are not using PL/I or one of the languages supported by a preprocessor.

If you are combining Callable RDO and preprocessed RDBPRE or RDML statements, declare the RDB$_ symbolic error codes only once.

19.7.2.3 Calling LIB$MATCH_COND When you want to determine which of several errors has caused a call to RDB$INTERPRET to fail, you can use the VMS Run-Time Library routine LIB$MATCH_COND.

The LIB$MATCH_COND routine compares the first argument in its argument list to the remaining arguments. If a match is found, it returns the position in the argument list of the matching argument. If no match is found, LIB$MATCH_COND returns a zero.

You could evaluate the return status value directly with your programming language's SELECT, CASE, EVALUATE, or IF statement, without calling the LIB$MATCH_COND routine. However, future versions of Rdb/VMS may change the severity levels or facility names of certain symbolic error codes. If this were to happen, you would have to link your program again under the new version so that the program would detect the correct error codes. The LIB$MATCH_COND routine matches only the condition identifier of the return status value and is unaffected by changes in severity levels or facility names. For this reason, you should use the LIB$MATCH_COND routine.

Example 19–13, from the HANDLE_ERROR procedure, demonstrates how to call the LIB$MATCH_COND routine in a PL/I Callable RDO program. Declare LIB$MATCH_COND as an EXTERNAL ENTRY in PL/I.

Example 19–13 Error Handling in Callable RDO

```
HANDLE_ERROR: PROCEDURE;

/**********************************************************/
/* This procedure handles run-time errors detected by the   */
/* ON ERROR clause in the Callable RDO programs.            */
/**********************************************************/

%REPLACE SECONDS_TO_WAIT BY 5;

DECLARE error                 FIXED BINARY (15);
DECLARE string                CHARACTER (132);
DECLARE msg_string            CHARACTER (132);
DECLARE error_len             FIXED BINARY (15);
DECLARE lock_error            BIT (1);

msg_string = ' ';
succeed      = '0'B;


/* Use LIB$MATCH_COND to determine which of a series of */
/* errors might have occurred.                          */

error = LIB$MATCH_COND (RDB_STATUS,
                        ADDR (RDB$_DEADLOCK),
                        ADDR (RDB$_LOCK_CONFLICT),
                        ADDR (RDB$_NO_DUP),
                        ADDR (RDB$_NOT_VALID),
                        ADDR (RDB$_INTEG_FAIL),
                        ADDR (RDB$_STREAM_EOF),
                        ADDR (RDO$_DATCONERR),
                        ADDR (RDB$_NO_RECORD));


/* The SELECT statement directs the program to appropriate  */
/* statements to execute, depending on the error that       */
/* was detected.                                            */

SELECT;

/* Unexpected error */

  WHEN  (ERROR = 0) DO;
          OPEN FILE (err_file)  TITLE ('error_file');
          PUT SKIP LIST ('Unexpected error - terminating program');
          err = SYS$GETMSG(RDB_STATUS, ADDR (error_len),
                                DESCRIPTOR (msg_string), 0, 0);
          PUT SKIP FILE (err_file) LIST (msg_string);
          CALL RDB$SIGNAL();
          CLOSE FILE (err_file);
          END;
```

(continued on next page)

Example 19–13 (Cont.) Error Handling in Callable RDO

```
/* Deadlock or lock conflict */

  WHEN  (ERROR = 1, ERROR = 2) DO;
          IF (retry <= 4) THEN DO;
              PUT SKIP LIST ('Deadlock or Lock conflict error');
              PUT SKIP LIST ('Others are using the data that'!!
                             'you want to access');
              err = LIB$WAIT(SECONDS_TO_WAIT);
              END;
          ELSE DO;
              PUT SKIP LIST ('Sorry, resources are not available, ');
              PUT SKIP LIST ('please retry later');
              END;
          END;

/* Duplicates not allowed  */

  WHEN  (ERROR = 3) DO;
          PUT SKIP LIST ('Duplicates are not allowed');
          err = SYS$GETMSG(RDB_STATUS, ADDR (error_len),
                                DESCRIPTOR (msg_string), 0, 0);
          PUT SKIP LIST (msg_string);
          END;

/* Invalid data */

  WHEN  (ERROR = 4) DO;
          PUT SKIP LIST ('Invalid Data');
          err = SYS$GETMSG(RDB_STATUS, ADDR (error_len),
                                DESCRIPTOR (msg_string), 0, 0);
          PUT SKIP LIST (msg_string);
          END;

/* Integrity failure */

  WHEN   (ERROR = 5) DO;
          PUT SKIP LIST ('Integrity failure');
          err = SYS$GETMSG(RDB_STATUS, ADDR (error_len),
                                DESCRIPTOR (msg_string), 0, 0);
          PUT SKIP LIST (msg_string);
          END;

  WHEN   (ERROR = 6) DO;
          END;
```

(continued on next page)

Example 19–13 (Cont.) Error Handling in Callable RDO

```
/* Invalid date  */

  WHEN  (ERROR = 7) DO;
          PUT SKIP LIST ('Invalid Date');
          err = SYS$GETMSG(RDB_STATUS, ADDR (error_len),
                                  DESCRIPTOR (msg_string), 0, 0);
          PUT SKIP LIST (msg_string);
          END;

/* Record deleted */

  WHEN   (ERROR = 8) DO;
          PUT SKIP LIST ('A record entered during this session has');
          PUT SKIP LIST ('been deleted');
          END;

   OTHERWISE ERROR = ERROR;
   END;

END;
```

19.7.3 Displaying Error Messages in Callable RDO Programs

The method you choose to display error messages depends on several factors. If you want to:

- Display an error message generated by Rdb/VMS and optionally terminate your program, call the RDB$SIGNAL routine.
- Display an error message generated by Rdb/VMS and continue program execution, call the SYS$PUTMSG system service.
- Use an error message generated by Rdb/VMS within your program and continue program execution, call the SYS$GETMSG system service.
- Display user-supplied error messages, or a mixture of user-supplied error messages and Rdb/VMS error messages, call the SYS$GETMSG or SYS$PUTMSG system service with a user-defined error code.

Information on creating user-supplied error messages is contained in Chapter 10.

19.7.3.1 Calling RDB$SIGNAL Call the RDB$SIGNAL routine when you want to display an error message generated by Rdb/VMS and (optionally) terminate your program. RDB$SIGNAL is an Rdb/VMS routine that calls the LIB$SIGNAL routine with the Rdb/VMS message vector. The Rdb/VMS message vector is a signal argument vector that contains pointers to additional error messages and to the FAO arguments required to format the Rdb/VMS error messages. The Rdb/VMS message vector is not accessible to the Callable RDO programs without the use of the RDB$SIGNAL routine.

LIB$SIGNAL is a VMS Run-Time Library routine that:

- Receives the return status value of the error or exception
- Causes a signal condition, which causes the appropriate catchall condition handler to pass the signal argument vector to the SYS$PUTMSG system service

  The SYS$PUTMSG system service calls the SYS$GETMSG system service to retrieve the message, and then the SYS$FAO service formats the error message and displays it on your terminal (or in the batch log).
- Resignals the error to the traceback or catchall default condition handler

  If the error is not fatal, program execution continues. If the error is fatal, the host language error handler signals the error to the default condition handler, which terminates program execution.

In FORTRAN, Pascal, C, and any language that does not define its own condition handler, you can continue program execution after the call to the RDB$SIGNAL routine even when the error is fatal. See Example 19–16 in Section 19.7.5, for an example of using a condition handler to continue program execution despite a fatal error after a call to RDB$SIGNAL.

In BASIC, COBOL, and PL/I, use SYS$GETMSG instead of RDB$SIGNAL to continue program execution after a fatal error. See Example 19–15 in Section 19.7.5 for an example of continuing after a fatal error in FORTRAN.

The BASIC and FORTRAN format of the RDB$SIGNAL calling sequence is:

```
CALL RDB$SIGNAL()
```

The COBOL format of the RDB$SIGNAL calling sequence is:

```
CALL "RDB$SIGNAL".
```

The Pascal format of the RDB$SIGNAL calling sequence is:

```
RDB$SIGNAL;
```

The PL/I format of the RDB$SIGNAL calling sequence is:

```
CALL RDB$SIGNAL();
```

The C format of the RDB$SIGNAL calling sequence is:

```
rdb$signal():
```

Your method of declaring RDB$SIGNAL is language-specific; refer to your programming language user’s guide for this information if you are not using PL/I or a language supported by a preprocessor.

See an earlier example, Example 19–13, for an illustration of the use of the RDB$SIGNAL routine in a PL/I program that uses Callable RDO.

19.7.3.2 Calling SYS$GETMSG

You can call the SYS$GETMSG system service when you want to use an Rdb/VMS error message within your program, or to display an Rdb/VMS error message and continue program execution.

The first argument in the call to the SYS$GETMSG system service is the Rdb/VMS return status value, the unique identification for the Rdb/VMS error message. The SYS$GETMSG system service locates the error message and returns it to your program in the second argument of the call. You must declare a character string to receive the message. Your program can then manipulate this character string in any way it chooses—for example, it can:

- Display the string
- Write the string to a file
- Evaluate substrings within the string

See the section on calling the SYS$GETMSG system service in each of the language-specific chapters for more information on the format and use of this service.

The SYS$GETMSG system service does not format the FAO arguments in the error message it returns to your program. When you require formatted FAO arguments in the error message, use RDB$SIGNAL rather than SYS$GETMSG. In Callable RDO, the format parameters are not available to your program unless you call RDB$SIGNAL.

You could use the SYS$PUTMSG system service in a Callable RDO application when you want to display an error message generated by Rdb/VMS and continue program execution. The SYS$PUTMSG system service writes the error message to the terminal and to the error file, SYS$ERROR. You can direct SYS$ERROR at DCL level to your program error file when you want the SYS$PUTMSG service to write an Rdb/VMS error message to your error file.

However, the SYS$PUTMSG service expects the message vector, RDB$MESSAGE_VECTOR, which is only accessible through the RDB$SIGNAL routine. The SYS$PUTMSG system service will accept the return status value instead of the message vector, but in this case will be unable to format the FAO arguments in the error message unless you build your own signal argument vector. For this reason, unless you create your own signal argument vector, you should use SYS$PUTMSG in Callable RDO only if you also use RDB$SIGNAL. See Example 19–16 in Section 19.7.5 for a demonstration of the use of the SYS$PUTMSG system service with RDB$SIGNAL.

An earlier example, Example 19–13, illustrates the use of the SYS$GETMSG system service in a PL/I Callable RDO program.

19.7.4 Error Recovery

Error recovery is specific to the program in which the error occurs. Frequently the individual program logic requires an individual error routine. However, there are several categories of Rdb/VMS errors that occur in programs:

- Multi-user conflicts
- Integrity and constraint failures
- Fatal or unexpected errors

See Chapter 10 for a full explanation of these errors.

Additionally, Callable RDO does not provide end-of-stream condition handling automatically. When you fetch records within a stream or get records within a segmented string, your program must check the end-of-stream condition at each FETCH statement.

When processing segmented strings, you must check for the end-of-stream condition at the segmented string GET statement. When your error handler detects the end-of-stream exception condition, your program can close the stream or segmented string and proceed to the next logical operation. After starting a record stream using the START_STREAM statement, you point to successive records by using the FETCH statement. After retrieving the last record, the next FETCH statement returns the end-of-stream condition, RDB$_STREAM_EOF. Your program must explicitly detect this condition. When your error handler detects the RDB$_STREAM_EOF condition, it should call the RDB$INTERPRET function to close the stream and continue program execution as necessary.

If you are retrieving segmented strings in a Callable RDO program, you need to use two logic loops, each with its own stream. The outer loop forms the record stream and the inner loop forms the segmented string stream. You advance the pointer of the record stream in the outer loop using the FETCH statement, but you do not use this statement with the segmented string stream. The GET statement both advances the pointer and retrieves the segment. In retrieving segmented strings, your error handler must detect the end of the record stream (RDB$_STREAM_EOF) and the end of the segmented string stream (RDO$_NO_MORSEG). Your program logic proceeds as follows:

1. Fetch a record from the record stream.
2. Get a segment from the segmented string.
3. Continue to get segments until the end-of-segmented-string-stream condition is met.
4. Fetch another record. If end of the record stream, end; otherwise, go to Step 2.

In Callable RDO programs:

- To detect end-of-stream check for RDB$_STREAM_EOF
- To detect end-of-segmented-string check for RDO$_NO_MORSEG

Example 19–14, from the LIST_RECORD procedure, shows handling the end-of-stream condition in PL/I.

Example 19–14 Handling a Record Stream End Condition in Callable RDO

```
LIST_RECORD: PROCEDURE;

/**************************************************************************/
/* This procedure lists all the employees and the colleges they attended. */
/**************************************************************************/

  /* Declare variables. */

  DECLARE degree        CHARACTER(3);
  DECLARE degree_field  CHARACTER(15);
  DECLARE found         BIT;

  /* Initialize variables. */
  err          = 0;
  first_name   = ' ';
  last_name    = ' ';
  degree       = ' ';
  degree_field = ' ';

  /* Start transaction. */
  RDB_COMMAND = 'START_TRANSACTION READ_ONLY ';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;


  /* For each EMPLOYEES record that has a corresponding record in DEGREES, */
  /* print the DEGREES record.                                            */

  RDB_COMMAND = 'START_STREAM ES USING E IN '
              !! 'EMPLOYEES SORTED BY E.LAST_NAME';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

  FETCH_COMMAND = 'FETCH ES';
  RDB_COMMAND = 'GET !VAL = E.FIRST_NAME; '
               !! '!VAL = E.EMPLOYEE_ID; '
               !! '!VAL = E.LAST_NAME END_GET;';
```

(continued on next page)

Example 19–14 (Cont.) Handling a Record Stream End Condition in Callable RDO

```
RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (FETCH_COMMAND));

DO WHILE (RDB_STATUS_SUCCESS);
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND),
                             DESCRIPTOR (first_name),
                             DESCRIPTOR (employee_id),
                             DESCRIPTOR (last_name));

  found = '0'B;

  NRDB_COMMAND = 'START_STREAM DS USING D IN '
               !! 'DEGREES WITH D.EMPLOYEE_ID = !VAL';
  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND),
                DESCRIPTOR (employee_id));

  NFETCH_COMMAND = 'FETCH DS';
  NRDB_COMMAND = 'GET !VAL = D.DEGREE; '
                 !! '!VAL = D.DEGREE_FIELD END_GET;';

  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NFETCH_COMMAND));

  DO WHILE (NRDB_STATUS_SUCCESS);
    NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND),
                                DESCRIPTOR (degree),
                                DESCRIPTOR (degree_field));


    /* Print the names of the employees who have a record */
    /* stored in the DEGREES relation.                     */

    PUT SKIP LIST ('Name is: ', first_name, ' ', last_name);
    PUT SKIP LIST ('Degree is : ', degree);
    PUT SKIP LIST ('Degree field is: ',degree_field);

    found = '1'B;

    NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NFETCH_COMMAND));

  END;  /* Do NFETCH  */

  NRDB_COMMAND = 'END_STREAM DS';
  NRDB_STATUS = RDB$INTERPRET(DESCRIPTOR (NRDB_COMMAND));
  IF (^NRDB_STATUS_SUCCESS) THEN
      CALL HANDLE_ERROR;

  /* Print the records of the employees who do not have a record */
  /* stored in the DEGREES relation.                              */

  IF ^(found) THEN DO;
    PUT SKIP LIST (first_name, ' ', last_name);
    PUT SKIP LIST ('Does not have this information stored in the
                  database');
  END; /* if ^found */


  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (FETCH_COMMAND));

  END; /* Do FETCH */
```

(continued on next page)

Example 19–14 (Cont.) Handling a Record Stream End Condition in Callable RDO

```
    RDB_COMMAND = 'END_STREAM ES';
    RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

  RDB_COMMAND = 'ROLLBACK ';
  RDB_STATUS = RDB$INTERPRET(DESCRIPTOR (RDB_COMMAND));
      IF (^RDB_STATUS_SUCCESS) THEN
          CALL HANDLE_ERROR;

END LIST_RECORD;
```

19.7.5 Handling Fatal Errors

In some instances, the cause of fatal errors is located in the database, not the program. For example, your program may attempt to access a relation that has been deleted by the database administrator or the process that runs the program may not have sufficient privilege to modify a particular relation. There is little that your program can do to correct this type of error. However, your program can determine which fatal error has occurred, perform cleanup functions, display an error message, and optionally terminate the program.

In other cases, you can anticipate a fatal error and design an alternate logical path to which the program can branch if that error occurs. In this case, your program could do the following:

- Determine which error code was returned (using the LIB$MATCH_COND routine), to make sure it is the fatal error you expected.
- Call the SYS$PUTMSG or SYS$GETMSG system service to generate an error message.
- Perform any necessary database cleanup.
- Continue program execution along the alternate path.

You can establish your own condition handler that will permit your program to continue after a call to the RDB$SIGNAL routine. See Example 19–16 for the use of a condition handler to continue program execution despite a fatal error after a call to RDB$SIGNAL.

If you have detected a fatal error and you do not intend to continue program execution, you should perform whatever cleanup operations are necessary. The following is a list of typical cleanup operations:

- End streams
- Roll back transactions

- Finish Rdb/VMS databases
- Write an error message to a transaction audit file
- Close files

If you call the RDB$SIGNAL routine without establishing a condition handler, RDB$SIGNAL displays the error message and terminates your program. When using the RDB$SIGNAL routine, you should not perform any cleanup operations that involve Rdb/VMS statements. If you issue an Rdb/VMS statement after the error condition, the new statement alters the message vector that RDB$SIGNAL passes to LIB$SIGNAL, and you will not receive the correct error message. Instead, call the RDB$SIGNAL routine without executing any new calls to the RDB$INTERPRET function. When your program terminates after the call to the RDB$SIGNAL routine, the database monitor rolls back the transaction and performs the necessary database cleanup.

In Example 19–15, from the function GETSUPER in the DEPTFOR.FOR sample program, fatal errors other than deadlock, lock conflict, or duplicate index cause error message display and program termination.

Example 19–15 Handling Fatal Errors in Callable RDO

```
      INTEGER FUNCTION GETSUPER(ID)

C---------------------------------------------------------
C     This function returns -1 if successfully completed,
C     0 if deadlock or lock conflict occurs, 1 if there is
C     a program error.
C---------------------------------------------------------
      IMPLICIT NONE

      CHARACTER*(*) ID               !ID passed to function

      CHARACTER*5 SUPER_ID           !supervisor ID
      CHARACTER*5 SAVE_ID            !storage for supervisor ID
      CHARACTER*4 SUPER_DEPT         !supervisor's department
      CHARACTER*15 LAST_NAME         !supervisor's last name
      CHARACTER*200 RDB_STR          !string to hold RDO statements
      CHARACTER*15 ARG_1, ARG_2, ARG_3    !strings to pass arguments
                         .
                         .
                         .
```

(continued on next page)

Example 19–15 (Cont.) Handling Fatal Errors in Callable RDO

```
16    RDB_STR = 'get !val = c.supervisor-id;'//
     X          '!val = sc.last-name;'//
     X          '!val = sc.department-code end-get'
      ARG_NUM = 3
      OK = INTERPN(RDB_STR, SUPER_ID, LAST_NAME,
     X             SUPER_DEPT, ARG_NUM)
      IF (OK) 18, 30, 40
                       .
                       .
                       .
      INTEGER*4 FUNCTION INTERPN(RDO_STR, ARG_1, ARG_2, ARG_3, ARG_NUM)
C-------------------------------------------------------------------
C     This function calls RDB$INTERPRET with n !VAL arguments; it
C     returns -1 if the call was successful, 0 if deadlock
C     or lock conflict, 1 if dept-code already exists. If an un-
C     expected fatal error is detected, the function calls RDB$SIGNAL
C     and does NOT handle the error.
C-------------------------------------------------------------------
      IMPLICIT NONE

      CHARACTER*(*) RDO_STR               !RDO string passed to function
      CHARACTER*(*) ARG_1, ARG_2, ARG_3    !arguments passed to function
      INTEGER*4 ARG_NUM          !number of arguments passed to function

      INTEGER*4 STAT             !return status for call to RDB$INTERPRET
      INTEGER*4 ERR              !variable returned by LIB$MATCH_COND
      INTEGER*4 RDB$INTERPRET    !data type for function RDB$INTERPRET
      INTEGER*4 LIB$MATCH_COND   !data type for function LIB$MATCH_COND
      CHARACTER*80 MSG_STR       !string to receive error message
                       .
                       .
                       .
C-------------------------
C     begin function logic
C-------------------------
      IF (ARG_NUM .EQ. 1) THEN        !call interpreter with 1 argument

         STAT = RDB$INTERPRET(%DESCR(RDO_STR),
     1                        %DESCR(ARG_1))

      ELSE IF (ARG_NUM .EQ. 2) THEN   !call interpreter with 2 arguments

         STAT = RDB$INTERPRET(%DESCR(RDO_STR),
     1                        %DESCR(ARG_1),
     2                        %DESCR(ARG_2))

      ELSE IF (ARG_NUM .EQ. 3) THEN   !call interpreter with 3 arguments

         STAT = RDB$INTERPRET(%DESCR(RDO_STR),
     1                        %DESCR(ARG_1),
     2                        %DESCR(ARG_2),
     3                        %DESCR(ARG_3))

      ELSE
```

(continued on next page)

Example 19–15 (Cont.) Handling Fatal Errors in Callable RDO

```
         WRITE (3, 1) ARG_NUM         !write error message to error file
         WRITE (5, 1) ARG_NUM           !write error message to terminal

      END IF

      IF ((STAT .AND. 1) .NE. 0) THEN   !call was successful

         INTERPN = -1
         RETURN                         !continue main module logic

      ELSE

         ERR = LIB$MATCH_COND(%REF(STAT),
     1                  %LOC(RDB$_LOCK_CONFLICT),
     2                  %LOC(RDB$_DEADLOCK),
     3                  %LOC(RDB$_NO_DUP),
     4                  %LOC(RDB$_INTEG_FAIL),
     5                  %LOC(RDB$_UNRES_REL),
     6                  %LOC(RDB$_READ_ONLY_VIEW),
     7                  %LOC(RDB$_NO_CUR_REC),
     8                  %LOC(RDB$_NO_RECORD),
     9                  %LOC(RDB$_REQ_NO_TRANS))

      END IF

      GO TO (10,20,30,40,40,40,40,40), ERR    !handle expected errors

C---------------------------------------
C     ERR equals 0, an unexpected error
C---------------------------------------

      CALL SYS$GETMSG(%REF(STAT),,%DESCR(MSG_STR))
      WRITE (3, 2) MSG_STR       !write error message to error file
      WRITE (5, 3)               !write to terminal
      CALL RDB$SIGNAL()          !send errors to terminal and quit
                   .
                   .
                   .
C-------------------------------------
C     Other expected but fatal errors
C-------------------------------------

40    CALL SYS$GETMSG(%REF(STAT),,%DESCR(MSG_STR))
      WRITE (3, 6) MSG_STR       !write error message to error file
      WRITE (5, 7)               !write message to terminal
      CALL RDB$SIGNAL()          !send errors to terminal and quit


1     FORMAT ('0','Program error, ARG_NUM = ', I6)

2     FORMAT ('0','Unexpected fatal RDB$INTERPRET error,
     X terminating DEPTFOR.FOR'/ A80)
3     FORMAT ('0','Unexpected fatal RDB$INTERPRET error,
     X terminating DEPTFOR.FOR')
```

(continued on next page)

Example 19–15 (Cont.) Handling Fatal Errors in Callable RDO

```
4     FORMAT ('0','Lock conflict, rolling back transaction')
5     FORMAT ('0','Deadlock, rolling back transaction')

6     FORMAT ('0','Expected fatal RDB$INTERPRET error,
     X terminating DEPTFOR.FOR'/ A80)
7     FORMAT ('0','Expected fatal RDB$INTERPRET error,
     X terminating DEPTFOR.FOR')

      END
```

If your programming language does not provide its own error handling, you can create your own condition handler. You can use the VMS Run-Time Library routine LIB$ESTABLISH (or VAXC$ESTABLISH in C), to create a condition handler that allows your program to continue after calling the RDB$SIGNAL or LIB$SIGNAL routine with a fatal error. In Pascal, use the Pascal function ESTABLISH to create your own condition handler.

When an error occurs, the RDB$INTERPRET function does not signal the error. Instead, it returns an error condition in the return status value. Your program error handler evaluates the return status value and decides to continue or terminate program execution. However, when you call the RDB$SIGNAL or LIB$SIGNAL routine, the fatal error condition is *signaled*. Unless you establish a condition handler to handle the error, the fatal error condition is resignaled through all existing condition handlers and the program terminates.

This section discusses fatal error handling only in those programming languages that do not define their own condition handling; that is, this section does not apply to BASIC, COBOL, and PL/I. If your programming language is BASIC, COBOL, PL/I, or any other language that establishes its own condition handler, and you call the LIB$ESTABLISH routine, your programming language's error handling is disabled. For this reason, you should not use the LIB$ESTABLISH routine with such languages. Instead, call the SYS$PUTMSG or SYS$GETMSG system service, which does not signal the error. Your program can evaluate the error and perform whatever operations are necessary to continue after the fatal error.

When an error is signaled during program execution, the system condition handling facility passes control to the most immediate condition handler. The LIB$ESTABLISH routine establishes your condition handler as the most immediate condition handler. When the RDB$SIGNAL or LIB$SIGNAL routine signals a fatal error, control passes to your condition handler.

Your condition handler can choose to resignal the error by returning the value SS$_RESIGNAL. Or, your handler can choose not to resignal the error by returning the value SS$_CONTINUE or SS$_UNWIND.

If your condition handler resignals (that is, returns the value SS$_RESIGNAL):

- Control passes to your programming language condition handlers farther down the stack.
- The condition handlers resignal the error to the system error handler.
- If the error is sufficiently severe, the program terminates.

If your condition handler does not resignal, but instead returns the value SS$_CONTINUE:

- Control returns to the program statement immediately following the call to the RDB$SIGNAL routine.
- Program execution continues.

If you choose not to resignal the error, your program logic should make sure that the error will not adversely affect continued program execution. (See the *VMS RTL Library (LIB$) Manual* for a complete description of the use of the LIB$ESTABLISH with LIB$SIGNAL routines, and see the VAX C documentation set for a complete description of the VAXC$ESTABLISH routine.)

The following FORTRAN program demonstrates the use of the LIB$ESTABLISH routine to create a condition handler. The condition handler, HANDLER, evaluates a COMMON flag, FATAL. If the flag FATAL is set to true (the error is fatal), HANDLER sets the low-order bits of the condition value to a severity level of 4 (FATAL) and resignals the error. If the flag FATAL is not true, HANDLER sets the low-order bits of the condition value to a severity level of 3 (INFORMATIONAL) and returns the value SS$_CONTINUE.

Example 19–16, from the DEPTFOR.FOR program, uses the INTERP0 function to call the RDB$INTERPRET function with no arguments. The INTERP0 function:

- Calls LIB$ESTABLISH to create the user-defined condition handler, HANDLER.
- Calls RDB$INTERPRET.
- Evaluates the return status value, STAT.
- Sets the flag FATAL to TRUE if the fatal error is unexpected.
- Sets the flag FATAL to FALSE if the fatal error is: “index or constraint already defined”.

- Calls RDB$SIGNAL for fatal errors other than deadlock and lock conflict. The RDB$SIGNAL routine displays the error message and resignals the error to HANDLER. This condition handler:
  - Sets the return status SIGARGS(2) severity level to 4 (FATAL) and returns SS$_RESIGNAL if the flag FATAL is TRUE.
  - Sets the return status SIGARGS(2) severity level to 3 (INFORMATIONAL) and returns SS$_CONTINUE if the flag FATAL is FALSE.
- Terminates program execution if HANDLER returns the value SS$_RESIGNAL.
- Continues program execution if HANDLER returns the value SS$_CONTINUE.

Example 19–16 Continuing Program Execution After a Fatal Error in Callable RDO

```
      INTEGER FUNCTION INTERP0(RDO_STR)

C-------------------------------------------------------------------
C     This function calls RDB$INTERPRET with no !VAL arguments.
C     It returns -1 if the call was successful, 0 if deadlock or
C     lock conflict is detected, 1 if stream EOF; if constraint or
C     index is already defined, the function sets the COMMON flag
C     FATAL to false, and calls RDB$SIGNAL which will signal to
C     the condition handler. If there is an unexpected fatal error,
C     the function sets the COMMON flag FATAL to true and calls
C     RDB$SIGNAL which will resignal to the condition handler.
C--------------------------------------------------------------------

      IMPLICIT NONE
      INCLUDE '($SSDEF)'

      CHARACTER*(*) RDO_STR      !RDO string passed to function

      LOGICAL*2 FATAL            !flag to set if unexpected fatal error
      INTEGER*4 STAT             !return status for call to RDB$INTERPRET
      INTEGER*4 ERR              !variable returned by LIB$MATCH_COND
      INTEGER*4 RDB$INTERPRET    !data type for function RDB$INTERPRET
      INTEGER*4 LIB$MATCH_COND   !data type for function LIB$MATCH_COND
      CHARACTER*80 MSG_STR       !string to receive error message

      EXTERNAL HANDLER           !condition handler
      EXTERNAL RDB$SIGNAL
      EXTERNAL LIB$ESTABLISH
      EXTERNAL LIB$MATCH_COND
      EXTERNAL RDB$INTERPRET
      EXTERNAL SYS$GETMSG

      COMMON FATAL               !make flag available to function HANDLER
```

(continued on next page)

Example 19–16 (Cont.) Continuing Program Execution After a Fatal Error in Callable RDO

```
C----------------------
C     Errors to handle:
C----------------------

      INTEGER*4 RDB$_LOCK_CONFLICT   !lock conflict
      INTEGER*4 RDB$_DEADLOCK        !deadlock
      INTEGER*4 RDO$_INDEXTS         !index already defined
      INTEGER*4 RDO$_CONALREXI       !constraint already defined
      INTEGER*4 RDB$_STREAM_EOF      !stream EOF

      EXTERNAL RDB$_LOCK_CONFLICT
      EXTERNAL RDB$_DEADLOCK
      EXTERNAL RDO$_INDEXTS
      EXTERNAL RDO$_CONALREXI
      EXTERNAL RDB$_STREAM_EOF

C-------------------------
C     Begin function logic
C-------------------------

      CALL LIB$ESTABLISH(HANDLER)       !establish condition handler
      STAT = RDB$INTERPRET(%DESCR(RDO_STR))     !call interpreter

      IF ((STAT .AND. 1) .NE. 0) THEN           !call was successful

         INTERP0 = -1
         RETURN                         !continue main module logic

      ELSE

         ERR = LIB$MATCH_COND(%REF(STAT),
     1                    %LOC(RDB$_LOCK_CONFLICT),
     2                    %LOC(RDB$_DEADLOCK),
     3                    %LOC(RDO$_INDEXTS),
     4                    %LOC(RDO$_CONALREXI),
     5                    %LOC(RDB$_STREAM_EOF))

      END IF

       GO TO (10,20,30,40,50), ERR      !handle expected errors

C--------------------------------------------
C     LIB$MATCH_COND returns 0, no match found:
C     set flag so HANDLER will not handle error,
C     call RDB$SIGNAL to print error and quit
C--------------------------------------------

      FATAL = .TRUE.           !unexpected fatal error
      CALL RDB$SIGNAL()

10    INTERP0 = 0              !lock conflict
      WRITE (3, 1)             !write message to error file
      WRITE (5, 1)             !write message to terminal
      RETURN

20    INTERP0 = 0              !deadlock
      WRITE (3, 2)             !write message to error file
      WRITE (5, 2)             !write message to terminal
      RETURN
```

(continued on next page)

Example 19–16 (Cont.) Continuing Program Execution After a Fatal Error in Callable RDO

```
C--------------------------------------------------
C     LIB$MATCH_COND returns 3, index already
C     defined: set flag so HANDLER will handle error,
C     call RDB$SIGNAL to print error and continue
C--------------------------------------------------

30    FATAL = .FALSE.          !index already defined
      CALL RDB$SIGNAL()        !write errors to terminal

      INTERP0 = 1              !return 1 to DEFINNDX
      WRITE (3, 3)             !write message to error file
      WRITE (5, 3)             !write message to terminal
      RETURN

C--------------------------------------------------
C     LIB$MATCH_COND returns 4, constraint already
C     defined: set flag so HANDLER will handle error,
C     call RDB$SIGNAL to print error and continue
C--------------------------------------------------

40    FATAL = .FALSE.          !constraint already defined
      CALL RDB$SIGNAL()        !write errors to terminal

      INTERP0 = 1              !return 1 to DEFINCON
      WRITE (3, 4)             !write message to error file
      WRITE (5, 4)             !write message to terminal
      RETURN

50    INTERP0 = 1              !stream EOF, return 1 to MAIN
      RETURN

1     FORMAT ('0','Lock conflict, rolling back transaction')
2     FORMAT ('0','Deadlock, rolling back transaction')
3     FORMAT ('0','Nonfatal error, index already defined')
4     FORMAT ('0','Nonfatal error, constraint already defined')

      END
                        .
                        .
                        .
      INTEGER*4 FUNCTION HANDLER(SIGARGS, MECHARGS)

      IMPLICIT NONE
      INCLUDE '($SSDEF)'

      INTEGER*4 SIGARGS(20), MECHARGS(5)
      LOGICAL*2 FATAL

      COMMON FATAL

      EXTERNAL SYS$PUTMSG

C----------------------------------------------------
C     If error is fatal, set condition code to severe
C     error and resignal; else print error to terminal,
C     set severity level to 3 and continue
C----------------------------------------------------
```

(continued on next page)

Example 19–16 (Cont.) Continuing Program Execution After a Fatal Error in Callable RDO

```
      PRINT *, 'In HANDLER, evaluate FATAL'

10    IF (FATAL) THEN

        SIGARGS(2) =  JIBCLR(SIGARGS(2), 0)
        SIGARGS(2) =  JIBCLR(SIGARGS(2), 1)
        SIGARGS(2) =  JIBSET(SIGARGS(2), 2)
        HANDLER = SS$_RESIGNAL

      ELSE

        CALL SYS$PUTMSG(SIGARGS)
        SIGARGS(2) =  JIBSET(SIGARGS(2), 0)
        SIGARGS(2) =  JIBSET(SIGARGS(2), 1)
        SIGARGS(2) =  JIBCLR(SIGARGS(2), 2)
        HANDLER = SS$_CONTINUE

      END IF

      RETURN

      END
```

A
Programming Reference Tables

This appendix contains the following tables:

- Table A–1 lists the commonly used Rdb/VMS symbolic error codes for data manipulation statements.
- Table A–2 lists the commonly used Rdb/VMS symbolic error codes for data definition statements. Refer to Appendix B of the *VAX Rdb/VMS RDO and RMU Reference Manual* for information about the location of the files that contain explanations of the RDO, RDB, and RDMS facility error messages.

Table A–1 Commonly Used Rdb/VMS Symbolic Error Codes for Data Manipulation

Rdb/VMS Statement	Likely Errors	Less Likely Errors
COMMIT	RDB$_LOCK_CONFLICT RDB$_DEADLOCK RDB$_INTEG_FAIL	RDB$_BAD_TRANS_HANDLE RDB$_NO_PRIV RDB$_REQ_NO_TRANS RDO$_STRNOTOPE†
CREATE_SEGMENTED_ STRING	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_BAD_SEGSTR_HANDLE RDB$_NO_PRIV RDB$_REQ_NO_TRANS RDB$_SEGSTR_NO_TRANS
DATABASE	None	RDB$_NO_PRIV
END_SEGMENTED_ STRING	None	RDB$_BAD_SEGSTR_HANDLE

†Callable RDO programs only.

(continued on next page)

Table A–1 (Cont.) Commonly Used Rdb/VMS Symbolic Error Codes for Data Manipulation

Rdb/VMS Statement	Likely Errors	Less Likely Errors
END_STREAM	None	RDO$_STRNOTOPE†
ERASE	RDB$_LOCK_CONFLICT RDB$_DEADLOCK RDB$_INTEG_FAIL	RDB$_READ_ONLY_FIELD RDB$_READ_ONLY_REL RDB$_READ_ONLY_TRANS RDB$_READ_ONLY_VIEW RDB$_NO_PRIV RDB$_REQ_NO_TRANS
FETCH	RDB$_STREAM_EOF RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_NO_CUR_REC RDB$_NO_PRIV RDB$_WRONUMARG RDO$_STRNOTOPE† RDB$_STROUTSCO†
FINISH	None	RDB$_BAD_DB_HANDLE RDB$_INTEG_FAIL RDB$_OPEN_TRANS
FIRST . . . FROM	RDB$_FROM_NO_MATCH	None
FOR	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_ARITH_EXCEPT RDB$_NO_PRIV RDB$_WRONUMARG
GET	RDB$_LOCK_CONFLICT RDB$_DEADLOCK RDB$_SEGMENT† RDB$_SEGSTR_EOF RDO$_NO_MORESEG†	RDB$_NO_CUR_REC RDB$_NO_PRIV RDB$_NO_RECORD RDB$_OBSOLETE_METADATA RDB$_REQ_NO_TRANS RDB$_SEGSTR_NO_OP RDB$_SEGSTR_NO_READ RDB$_WRONGNUMARG

†Callable RDO programs only.

(continued on next page)

Table A–1 (Cont.) Commonly Used Rdb/VMS Symbolic Error Codes for Data Manipulation

Rdb/VMS Statement	Likely Errors	Less Likely Errors
MODIFY	RDB$_LOCK_CONFLICT RDB$_DEADLOCK RDB$_INTEG_FAIL RDB$_NO_DUP RDB$_NOT_VALID RDB$_BAD_SEGSTR_ HANDLE	RDB$_ARITH_EXCEPT RDB$_NO_CUR_REC RDB$_NO_PRIV RDB$_NO_SEGSTR_CLOSE RDB$_OBSOLETE_METADATA RDB$_READ_ONLY_FIELD RDB$_READ_ONLY_REL RDB$_READ_ONLY_TRANS RDB$_READ_ONLY_VIEW RDB$_REQ_NO_TRANS RDB$_SEGSTR_NO_WRITE RDB$_WRONUMARG
ROLLBACK	None	RDB$_BAD_TRANS_HANDLE RDB$_REQ_NO_TRANS
START_SEGMENTED_ STRING	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_NO_PRIV RDB$_SEGSTR_NO_TRANS
START_STREAM	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_ARITH_EXCEPT RDB$_NO_PRIV RDB$_WRONUMARG RDO$_STALROPE†
START_TRANSACTION	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDB$_BAD_DB_HANDLE RDB$_BAD_TRANS_HANDLE RDB$_EXCESS_TRANS RDB$_NO_PRIV
STORE	RDB$_LOCK_CONFLICT RDB$_DEADLOCK RDB$_INTEG_FAIL RDB$_NO_DUP RDB$_NOT_VALID	RDB$_NO_PRIV RDB$_NO_SEGSTR_CLOSE RDB$_OBSOLETE_METADATA RDB$_READ_ONLY_REL RDB$_READ_ONLY_TRANS RDB$_READ_ONLY_VIEW RDB$_REQ_NO_TRANS RDB$_REQ_WRONG_DB RDB$_SEGSTR_NO_WRITE RDB$_UNRES_REL RDB$_WRONUMARG

†Callable RDO programs only.

Table A–2 Commonly Used Rdb/VMS Symbolic Error Codes for Data Definition

Rdb/VMS Statement	Likely Errors	Less Likely Errors
CHANGE RELATION	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_AGGNOTALL RDO$_ANYNOTALL RDO$_BAD_END_NAME RDO$_DBKNOTALL RDO$_FLDNOTDEF RDO$_HOWCHANOT RDO$_RELNOTDEF
DEFINE CONSTRAINT	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_CONALREXI
DEFINE FIELD	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_INDEXTS RDO$_BAD_END_NAME
DEFINE RELATION	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_AGGNOTALL RDO$_ANYNOTALL RDO$_BAD_END_NAME RDB$_DBKNOTALL RDO$_REL_EXISTS
DEFINE VIEW	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_BAD_END_NAME RDO$_FLDNOTCRS RDO$_FLDNOTDEF RDO$_RELNOTDEF
DELETE CONSTRAINT	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_CONNOTDEF
DELETE FIELD	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_FLDNOTDEF
DELETE INDEX	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_INDNOTDEF
DELETE RELATION	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_RELNOTDEF
DELETE VIEW	RDB$_LOCK_CONFLICT RDB$_DEADLOCK	RDO$_VIEWNOTDEF

Index