VAX TDMS Request and Programming Manual

Order No. AA-GS14B-TE

August 1986

This manual describes the TDMS Request Definition Utility. It explains how to create TDMS requests and invoke them from application programs.

OPERATING SYSTEM: **VMS**

MicroVMS

SOFTWARE VERSION: **VAX TDMS V1.6**

digital equipment corporation, maynard, massachusetts

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

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

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

Copyright © 1986 by Digital Equipment Corporation. All rights reserved.

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

The following are trademarks of Digital Equipment Corporation:

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

digital™

Contents

How to Use This Manual xiii

Technical Changes and New Features xvii

Part One: Creating Requests

1 Introduction to Requests

1.1 General Concepts of Requests 1-1
1.1.1 Using Form Definitions 1-2
1.1.2 Using Record Definitions 1-2
1.2 Using Request Instructions. 1-3
1.2.1 Using the FORM IS Instruction 1-4
1.2.2 Using the RECORD IS Instruction. 1-5
1.2.3 Using the DESCRIPTION Instruction and Comment Text . . . 1-6
1.2.4 Using the CLEAR SCREEN Instruction 1-6
1.2.5 Displaying the Form 1-6
1.2.5.1 Using the DISPLAY FORM Instruction 1-6
1.2.5.2 Using the USE FORM Instruction 1-7
1.2.6 Moving Data to and from the Form. 1-9
1.2.6.1 Using the INPUT TO Instruction 1-9
1.2.6.2 Using the OUTPUT TO Instruction. 1-11
1.2.6.3 Using the WAIT Instruction 1-12
1.2.7 Using Video Field Instructions 1-12
1.2.8 Using the END DEFINITION Instruction 1-13
1.3 Rules for Entering Request Instructions 1-13

2 Using the Request Definition Utility (RDU)

2.1 How RDU Uses the CDD 2-2
2.1.1 Setting RDU to Your CDD Directory 2-2
2.1.1.1 Defining CDD$DEFAULT. 2-2
2.1.1.2 Using the SET DEFAULT Command 2-3
2.1.2 Using the SHOW DEFAULT Command 2-3
2.1.3 Naming Requests and Specifying Forms and Records 2-3
2.2 Creating a Request. 2-5
2.2.1 Using the Interactive Method. 2-5
2.2.2 Using the File Method 2-6
2.2.3 Using a Command File - DCL or RDU Level 2-6
2.3 Correcting Errors. 2-7

2.4 Validating Requests 2-7
2.4.1 Changing the RDU Validation Option 2-8
2.4.2 Using the /[NO]STORE Qualifier 2-9
2.4.3 How RDU Validates a Request 2-10
2.5 Copying Requests in the CDD 2-10
2.6 Modifying a Request 2-11
2.7 Listing a Request 2-11
2.8 Deleting a Request 2-12
2.9 Exiting RDU 2-13

3 Mapping Between Form Fields and Record Fields

3.1 How the TDMS Mapping Instructions Work 3-2
3.2 How to Specify Fields in a Mapping Instruction 3-3
3.3 When to Use the %ALL Syntax 3-4
3.3.1 Using %ALL to Map an Entire Form 3-4
3.3.2 Using %ALL to Map Between a Form and a Larger Record . . . 3-6
3.3.3 Using %ALL to Map Between a Form and a Smaller Record . . . 3-7
3.4 When to Use Explicit Mapping Syntax 3-9
3.4.1 Explicitly Mapping Between a Form and a Record 3-9
3.4.2 Using Explicit Syntax to Map from One Field to Many Fields . . 3-10
3.4.3 Making Explicit References Unique 3-13
3.4.3.1 Using Group Field Names 3-15
3.4.3.2 Using the Record Name 3-17
3.5 Using the %ALL and Explicit Syntax in the Same Request 3-19
3.6 Mapping from a Form to a Group Record Field 3-21

4 Making Sure Your Request Mappings Are Correct

4.1 Rules for Mapping 4-1
4.2 Making Sure Record and Form Field Structures Are Compatible . . . 4-2
4.2.1 TDMS Record Field Structures 4-2
4.2.2 TDMS Form Field Structures 4-3
4.2.3 Compatible Form and Record Field Structures 4-3
4.3 Making Sure Field Data Types and Length Are Compatible 4-4
4.3.1 TDMS Form Field Data Types and Lengths 4-4
4.3.2 The Record Data Types to Which You Map 4-6
4.4 Creating Mappings Between Compatible Data Types 4-7
4.5 Form Definition Listings 4-10
4.5.1 How to List the Form Definition 4-11
4.5.2 What You Need to Know About Form Definitions 4-11
4.6 Record Definition Listings 4-12
4.6.1 How to List the Record Definition 4-12
4.6.2 What You Need to Know About Record Definitions 4-12

5 Finding and Correcting Your Errors

5.1 Syntax Errors Found by RDU 5-1
5.2 Semantic Errors Found by RDU 5-3
5.2.1 Mapping Errors 5-3
5.2.1.1 %ALL Warning and Information Messages 5-3
5.2.1.2 Explicit Mappings and Error and Warning Level Messages 5-4
5.2.2 Other Semantic Errors Found by RDU 5-5
5.3 Semantic Errors Not Found by RDU 5-6
5.3.1 Order Execution Errors 5-6
5.3.2 Mapping Errors 5-7
5.3.3 Form-Related Errors 5-7
5.4 Correcting Errors. 5-8
5.4.1 Using the EDIT Command 5-8
5.4.2 Using the MODIFY Command 5-8
5.4.3 Defining RDU$EDIT 5-8
5.4.4 Using the SAVE Command 5-9

6 Using Conditional Instructions in Requests

6.1 Using Conditional Instructions. 6-1
6.2 Conditional Requests 6-2
6.3 Using Conditional Requests 6-3
6.4 How TDMS Executes a Conditional Instruction at Run Time. 6-5
6.4.1 Specifying Control Values 6-6
6.4.1.1 Specifying More Than One Conditional Instruction 6-8
6.4.1.2 Using Nested CONTROL FIELD IS Instructions 6-8
6.4.2 Specifying Case Values 6-9
6.4.2.1 Using the NOMATCH Case Value. 6-9
6.4.2.2 Using the ANYMATCH Case Value. 6-10
6.4.2.3 Conditional Use of Forms 6-11
6.4.2.4 Case Values When You Use More Than One Control Value 6-12
6.4.3 Match Instructions in a CONTROL FIELD IS Instruction. . . . 6-12

7 Mapping Between Form Arrays and Record Arrays

7.1 What Is an Array? 7-1
7.1.1 Types of Form Arrays You Can Map to and from. 7-2
7.1.2 Types of Record Arrays You Can Map to and from. 7-4
7.2 Syntax for Mapping Between Form and Record Arrays 7-6
7.2.1 Explicit Syntax for Mapping Array Elements 7-7
7.2.2 %ALL Syntax for Mapping Array Elements 7-8

7.3 Rules for Mapping Arrays 7-9
7.3.1 Explicit Mappings and Errors 7-10
7.3.2 %ALL Mappings and Errors. 7-10
7.4 Examples of Mapping Indexed and Scrolled Arrays 7-11
7.4.1 Explicit Mapping of Scrolled or Indexed Arrays 7-12
7.4.2 %ALL Mappings 7-14
7.4.2.1 %ALL Mapping and a Scrolled Array 7-14
7.4.2.2 Using %ALL Mappings and Indexed Arrays 7-15
7.4.3 Explicitly Mapping a Subset of a Scrolled or Indexed Array . . . 7-17
7.4.4 Mapping Scrolled Arrays to Several Record Arrays 7-18
7.4.4.1 Explicitly Mapping Several Scrolled Arrays 7-18
7.4.4.2 %ALL Mapping of Several Scrolled Arrays 7-20

8 Advanced Mapping Between Arrays

8.1 Horizontally-Indexed Scrolled Form Arrays 8-1
8.2 Two-Dimensional Record Arrays. 8-2
8.3 Syntax for Mapping Two-Dimensional Arrays. 8-4
8.4 General Rules for Two-Dimensional Arrays 8-4
8.5 Examples of Mapping Two-Dimensional Arrays 8-5
8.5.1 Explicit Syntax to Map a Two-Dimensional Array. 8-5
8.5.2 Rules for %ALL Mapping of Two-Dimensional Arrays 8-7
8.5.3 %ALL to Map a Two-Dimensional Array 8-7
8.5.4 Mapping a Subset of a Two-Dimensional Array. 8-9

9 Using an Array as a Control Value

9.1 How to Use an Array As a Control Value to Collect Varying Elements 9-1
9.2 How TDMS Evaluates a Control Value Array at Run Time 9-2
9.3 Rules for Specifying the Control Value Array 9-4
9.3.1 Explicitly Assigning Values to %LINE and %ENTRY 9-6
9.3.2 Using a Work Array as a Control Value Array 9-6
9.3.3 Specifying an Entire Array as a Control Value 9-6
9.4 Example - Using a One-Dimensional Control Value Array. 9-7
9.5 Example - Using a Two-Dimensional Control Value Array. 9-10

10 How to Display and Input Data in a Scrolled Region

10.1 How TDMS Displays and Collects Data in a Scrolled Array 10-2
10.2 How to Allow the Operator to View Data in Scrolled Regions 10-5

11 Program Request Keys

11.1 What Are Program Request Keys? 11-2
11.2 Using Program Request Keys. 11-4
11.3 Creating a Request That Uses a Program Request Key. 11-4
11.3.1 Default CHECK Mode Modifier 11-6
11.3.2 NO CHECK Modifier 11-8
11.4 Examples of Using Program Request Keys. 11-8
11.4.1 Using PRKs to Allow the Operator to Control Application Flow 11-8
11.4.2 Using a PRK to Return a Value to a Control Value 11-10

Part Two: Creating Request Libraries

12 Working with Request Libraries

12.1 Creating a Request Library Definition. 12-1
12.2 Copying a Request Library Definition 12-2
12.3 Listing a Request Library Definition. 12-2
12.4 Modifying a Request Library Definition. 12-3
12.5 Validating a Request Library Definition. 12-3
12.6 Deleting a Request Library Definition. 12-4
12.7 Building a Request Library File. 12-4

Part Three: Writing Application Programs

13 Introduction to TDMS Programming

13.1 TDMS Programming Calls 13-1
13.2 General Format of TDMS Programming Calls. 13-2

14 Using the Primary TDMS Synchronous Calls

14.1 Opening a Request Library File - TSS$OPEN_RLB 14-2
14.2 Opening a Channel - TSS$OPEN. 14-3
14.3 Transferring Data and Displaying Forms - TSS$REQUEST 14-3
14.4 Closing the Request Libary File - TSS$CLOSE_RLB 14-5
14.5 Closing a Channel - TSS$CLOSE 14-6
14.6 Testing the Return Status Code 14-6
14.7 Compiling and Linking a TDMS Program 14-9
14.7.1 Compiling a TDMS Program. 14-9
14.7.2 Linking a TDMS Program 14-9
14.8 Two Simple TDMS Programs. 14-10

15 Using Supplementary Calls

15.1 Reading Messages from the Terminal - TSS$READ_MSG_LINE . . 15-1

15.2 Sending Messages to the Terminal. 15-2

15.2.1 Writing to the Message Line - TSS$WRITE_MSG_LINE . . . 15-3

15.2.2 Interrupting a Request or an Existing Message Line Operation - TSS$WRITE_BRKTHRU 15-3

15.3 Copying the Current Form to a Specific Device - TSS$COPY_SCREEN 15-4

15.4 Canceling Input/Output Calls in Progress - TSS$CANCEL 15-6

16 Using Record Definitions

16.1 Using CDD Record Definitions in BASIC programs 16-2

16.1.1 Referring to a CDD Record Definition in BASIC 16-2

16.1.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax. 16-4

16.1.3 Referring to CDD Array Record Definitions in BASIC 16-5

16.2 Using COBOL to Refer to CDD Record Definitions. 16-7

16.2.1 Referring to a CDD Record Definition in COBOL 16-8

16.2.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax. 16-10

16.2.3 Referring to CDD Array Record Definitions in COBOL. 16-11

16.3 Using FORTRAN to Refer to CDD Record Definitions. 16-13

16.3.1 Referring to a CDD Record Definition in FORTRAN 16-14

16.3.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax. 16-15

16.3.3 Referring to CDD Array Record Definitions in FORTRAN. . . 16-16

16.4 Using Record Definitions Created by Database Management Systems 16-18

16.4.1 Using Record Definitions Created by VAX Rdb/VMS 16-19

16.4.2 Using Record Definitions Created by VAX DBMS. 16-20

16.4.3 Displaying and Updating Database Records in Scrolled Regions 16-22

16.4.3.1 Defining an Array Record 16-23

16.4.3.2 Declaring the Array Record in the Application and TDMS Requests 16-24

16.4.3.3 Creating a Collection of Records and Loading the Array . 16-24

16.4.3.4 Passing the Array to the Request. 16-25

16.5 Summary of Supported Data Types for Different Languages 16-25

17 Debugging a TDMS Application Program

17.1 How to Enable the Trace Facility 17-1
17.1.1 Defining a Logical Name 17-2
17.1.2 Issuing Trace Calls from an Application Program 17-2
17.2 Results of Using Trace 17-3
17.3 Debugging an Application Using Two Terminals 17-6

18 Application Function Keys (AFKS)

18.1 What Are Application Function Keys?. 18-1
18.2 When Do You Use Application Function Keys? 18-1
18.3 Declaring Application Function Keys 18-2
18.3.1 Terminal Keys You Can Declare as AFKs 18-3
18.3.2 How to Write an AST Routine 18-4
18.4 Removing an AFK Key Definition 18-4

19 Using Asynchronous Calls

19.1 What Are Asynchronous Calls?. 19-1
19.2 When Do You Use Asynchronous Calls? 19-2
19.3 The General Format for Asynchronous Calls 19-2

Figures

1-1 Request Format. 1-4
1-2 Mapping Between a Record Definition and a Form Definition 1-10
2-1 Suggested TDMS Design Sequence: Effects of Validation 2-9
4-1 Record Field Structures 4-2
4-2 Form Field Structures 4-3
6-1 A Conditional Request 6-3
6-2 Request Containing a CONTROL FIELD IS Instruction 6-4
6-3 How a Conditional Request Works. 6-7
7-1 Definition of an Array 7-2
7-2 Indexed Array. 7-3
7-3 Scrolled Array. 7-4
7-4 Simple One-Dimensional Record Arrays 7-5
7-5 One-Dimensional Group Arrays 7-5
7-6 Example of Mapping Arrays 7-6
7-7 The Underlying Form Array 7-11
7-8 Explicitly Mapping an Entire Scrolled Form Array. 7-13
7-9 Using %ALL to Map an Entire Scrolled Array. 7-14
7-10 Using %ALL to Map Entire Indexed Arrays 7-16
7-11 Mapping a Subset of an Indexed or Scrolled Array 7-17
7-12 Explicit Mapping of Several Scrolled Arrays 7-18
7-13 %ALL Mapping of Several Scrolled Arrays. 7-20

8-1 Horizontally-Indexed Scrolled Form Array 8-1
8-2 Using Explicit Syntax to Map a Two-Dimensional Array 8-6
8-3 Using %ALL Syntax to Map Two-Dimensional Arrays 8-8
8-4 Mapping a Subset of a Two-Dimensional Array 8-9
9-1 Using an Array as a Control Value 9-2
9-2 How TDMS Evaluates a Control Value Array at Run Time 9-3
9-3 The Scope of a Dependent Range 9-4
9-4 Illegal Nested Dependent Ranges 9-5
9-5 Specifying Nonarray or Single-Element Control Values 9-5
9-6 Explicitly Assigning Values to %LINE and %ENTRY. 9-6
9-7 Collecting Elements from Several Scrolled Fields 9-7
9-8 Using a Two-Dimensional Array as a Control Value. 9-11
10-1 Displaying Data in a Scrolled Region 10-6
11-1 A Sample PRK Instruction 11-3
11-2 Defining Program Request Keys 11-5
11-3 Using the CHECK Modifier. 11-7
11-4 Using PRKs to Allow Operator Control of Application Flow 11-9
11-5 Using PRKs in Conditional Instructions 11-10
14-1 Request That Displays a Form 14-10
16-1 Referring to a CDD Record Definition in BASIC 16-2
16-2 Referring to a CDD Record Definition Containing the VARIANTS Syntax in BASIC. 16-4
16-3 Referring to a CDD Record Definition with Nested Arrays 16-5
16-4 Referring to a Two-Dimensional CDD Array Record Definition in BASIC 16-7
16-5 Referring to a CDD Record Definition in COBOL 16-9
16-6 Referring to a CDD Record Definition with the VARIANTS Syntax in COBOL 16-10
16-7 Referring to a CDD Record Definition with Nested OCCURS Syntax in COBOL 16-11
16-8 Referring a Two-Dimensional CDD Array Record Definition in COBOL 16-12
16-9 Referring to a CDD Record Definition in FORTRAN 16-14
16-10 Referring to a CDD Record Definition Containing the VARIANTS Syntax in FORTRAN 16-15
16-11 CDDL Record Definition with Nested OCCURS Syntax 16-17
16-12 Referring to a Two-Dimensional CDD Array Record Definition in FORTRAN 16-17
17-1 Sample Trace Output. 17-3

Tables

4-1 Effect of Scale Factor on Form Field Data 4-5
4-2 TDMS Form Field Pictures and Form Field Data Types 4-5
4-3 Record Field Data Types 4-7
4-4 Simplified Compatible Input Mappings (Form Fields to Record Fields) 4-8
4-5 Simplified Compatible Output Mappings (Record Fields to Form Fields) 4-9
11-1 Run-Time Function Keys 11-1
16-1 Data Type Conversion Chart 16-26
18-1 Application Function Key Codes 18-3

How to Use This Manual

This manual describes the use of requests in the VAX Terminal Data Management System (TDMS) software, also referred to in this manual simply as TDMS. There are three parts to the manual, representing the three distinct steps for using requests:

- Creating requests using the Request Definition Utility (RDU)
- Creating and building request libraries using RDU
- Creating an application program that invokes the requests

Intended Audience

This manual is intended for TDMS users who need to describe terminal input and output (I/O) for an application program. You should:

- Be familiar with application design using screen forms
- Know how to store and access definitions in the VAX Common Data Dictionary
- Be familiar with the VMS operating system, the Digital Command Language (DCL), and a DIGITAL editor such as EDT

If you are new to TDMS, or are unfamiliar with the general concepts of forms and requests, you should read Chapters 1 and 2 of the *VAX TDMS Forms Manual* before proceeding with this manual.

If you are new to the VMS operating system, you should read the *Introduction to VAX/VMS* manual that comes with the VMS documentation set.

Operating System Information

To verify which versions of your operating system are compatible with this version of VAX TDMS, check the most recent copy of the *VAX System Software Order Table/Optional Software Cross Reference Table,* SPD 28.98.xx.

Structure

This manual has three parts: creating requests, creating request libraries, and writing application programs. There are 19 chapters and an index:

Part One: Creating Requests

Chapter 1	Provides an introduction to requests and describes the syntax for the most common request instructions.
Chapter 2	Explains how to use the Request Definition Utility (RDU) to create, change, and delete requests.
Chapter 3	Describes the request instructions that move data to and from a form.
Chapter 4	Explains how to make sure that the requests you create are correct.
Chapter 5	Explains RDU error messages and how to correct the errors that cause them.
Chapter 6	Introduces the concept of conditional requests and explains how they are used and how to create conditional requests using a control field.
Chapter 7	Explains how to transfer data between one-dimensional form and record arrays.
Chapter 8	Explains how to transfer data between two-dimensional form and record arrays.
Chapter 9	Explains how to use conditional instructions to collect data selectively in a scrolled field.
Chapter 10	Explains how TDMS displays and collects data in scrolled fields.
Chapter 11	Explains how to create requests that redefine terminal keys for special functions.

Part Two: Creating Request Libraries

Chapter 12	Introduces the concept of request libraries and explains how to create them in RDU.

Part Three: Writing Application Programs

Chapter 13	Introduces the concept of TDMS routine calls that open request libraries and invoke requests.
Chapter 14	Explains how to use the primary synchronous calls, and gives a simple example program in both VAX BASIC and VAX COBOL.
Chapter 15	Explains how to use the supplementary synchronous calls.
Chapter 16	Explains how to declare CDD records in application programs.
Chapter 17	Explains how to debug a TDMS application program.
Chapter 18	Explains how to define special functions for terminal keys using a TDMS routine call.
Chapter 19	Introduces the concept of asynchronous calls, and explains how to use them.

Related Documents

As you use this book, you may find the following manuals helpful:

VAX TDMS Forms Manual

VAX TDMS Reference Manual

VAX/VMS DCL Dictionary

VAX Common Data Dictionary Data Definition Language Reference Manual

VAX Common Data Dictionary Utilities Reference Manual

VAX Run-Time Library Reference Manual

VAX/VMS System Services Reference Manual

Conventions

This section explains the special symbols used in this book:

Symbol	Meaning
[ ]	In syntax diagrams, square brackets enclose optional items from which you can choose one or none. Square brackets also enclose array subscripts.
{ }	Braces enclose items from which you must choose one and only one alternative.
{| |}	Bars in braces indicate that you must choose one or more of the items enclosed.
...	Horizontal ellipses indicate that you can repeat the previous item one or more times.
. . .	Vertical ellipses in an example indicate that information unrelated to the example has been omitted.
( )	In RDU syntax, matching parentheses enclose lists of receiving fields in mapping instructions and CDD passwords.
RDU >	The RDU > prompt indicates the utility is at command level and ready to accept RDU commands.
RDUDFN >	Indicates that the RDU utility is at the instruction level and ready to accept request or request library instructions.
UPPERCASE	An uppercase word indicates a command or instruction keyword. Keywords are required unless otherwise indicated. Do not use keywords as variable names.
$	The dollar sign prompt indicates that you are at DIGITAL Command Language (DCL) level. (It is possible to change the DCL prompt. However, in this manual the examples use the default prompt, the dollar sign.)
CTRL/x	Indicates that you press both the CTRL (control) key and the specified key simultaneously.
Color	Colored text in examples shows what you enter.
<RET>	Indicates the RETURN key. Unless otherwise stated, end all input by pressing the RETURN key.

Technical Changes and New Features

This section summarizes the changes to VAX TDMS that are described in this manual.

- Information about using the ANYMATCH and NOMATCH instructions in conditional requests has been added to Chapter 6, Using Conditional Instructions in Requests.
- In addition to its regular date formats, TDMS now includes the standard VMS date format:

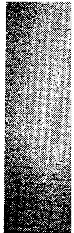
  DD-MMM-YYYY

  This information has been added to the Data Type Conversion Chart.

  For additonal information about this date format, see the Input and Output Mapping Tables in the *VAX TDMS Reference Manual.*
- TDMS has a predefined set of run-time function keys that operators can use to perform various operations on the screen such as moving from field to field, refreshing the screen, and getting help. These predefined keys are listed in Table 11-1 in the chapter called Program Request Keys.
- TDMS can now be used with VT200-series terminals set to VT200 mode. The following function keys from the LK201 keyboard used by VT200-series terminals are supported:
  - The F12 (BS) key performs the BACK SPACE key function.
  - The F13 (LF) key performs the LINE FEED key function.
  - The HELP key performs the PF2 or HELP key function.

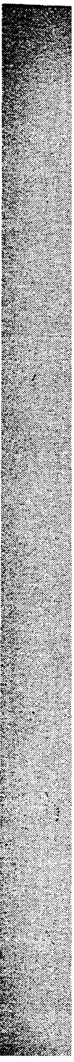
Information about these keys has been included where appropriate throughout the documentation set.

- The file names for the /OUTPUT and /LIST qualifiers to RDU commands now accept file names up to 39 characters in length, including dollar signs ($) and underscores (_).



Part One:

Insert first divider with text: Creating Requests



Introduction to Requests 1

This manual explains how to create TDMS requests and request libraries and invoke them from application programs. Requests are the central feature of VAX TDMS. If you are unfamiliar with the general concepts of VAX TDMS, you should read Chapters 1 and 2 of the *VAX TDMS Forms Manual* for an overview of the product before continuing with this manual. This chapter introduces the concept of requests and describes the syntax of a simple request.

1.1 General Concepts of Requests

A request is a set of instructions that you define outside of the application program. These instructions describe the exchange of data between a form and a TDMS application program.

When an application program calls a request at run time, the request can perform a variety of functions, including:

- Display a form
- Allow data to be entered on a form and returned to one or more records
- Display data from one or more records on a form
- Perform conditional operations, moving data between a form and a record depending on conditions within a program or on data entered by the operator
- Override certain features of a form definition (video attributes) without redefining the form
- Notify the program or operator of special conditions by displaying messages on the screen or in a record field

Because you describe all the terminal I/O for an application in one or more requests, you considerably reduce the amount of program code needed in any application. You also increase the independence of the application program from the forms and records that the application uses. This independence makes TDMS applications easier to develop and to maintain.

You use the Request Definition Utility (RDU) to create, change, or delete requests. RDU stores the requests in a central storage facility, the Common Data Dictionary (CDD). Chapter 2 describes the RDU commands you use to create, modify, and delete requests.

This chapter describes what a request looks like. First, however, you should understand how requests use:

- Form definitions
- Record definitions

1.1.1 Using Form Definitions

Requests use **form definitions** to format data on the terminal screen. Forms are created in the Form Definition Utility (FDU) and they define fields that the request uses for data entry and/or display. Since you refer to these fields when you move data between a form and a record in a request, you must know:

- The name of the form definition in the CDD
- The names of the fields
- The data types, lengths, and other characteristics of the form fields

You learn more about how to access and use form definitions in Chapter 4, Making Sure Your Request Mappings Are Correct. For information about creating forms, see the *VAX TDMS Forms Manual*.

1.1.2 Using Record Definitions

Requests use **record definitions** to send and receive data to the form fields. You can define a record definition using any of the following products:

- The Common Data Dictionary Data Definition Language (CDDL)
- VAX DATATRIEVE
- VAX DBMS
- VAX Rdb/VMS

The record definitions are then stored in the CDD. To use the definitions in a request, you must know:

- The names of the record definitions in the CDD
- The names of the record fields
- The data types, lengths, and other characteristics of the record fields

Note

For the purpose of examples, all the record definitions in this manual are presented in CDDL syntax.

You learn more about accessing and using record definitions in Chapter 4, Making Sure Your Request Mappings Are Correct. To learn how to create record definitions using CDDL, see the *VAX Common Data Dictionary Data Definition Language Reference Manual*.

1.2 Using Request Instructions

Once you define your forms and records, you are ready to create a request. Figure 1-1 contains an example of a simple TDMS request that uses the basic set of request instructions. The instructions are easy-to-learn, English-like statements.

In a request, you name the form definition and the record definition that you want to use. You also identify the form that you want to display on the terminal screen and the form and record fields between which you want to map (move) data.

There are two parts to a simple request:

- The header, which identifies the forms and records that the request uses.
- The base, which contains instructions that TDMS performs every time an application program calls this request. These instructions define how the form appears on the screen and which fields to use for input and output mapping.

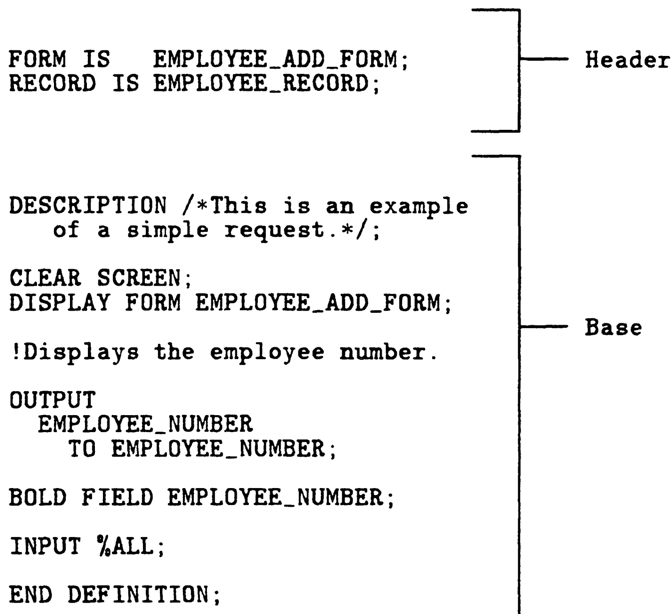


Figure 1-1: Request Format

The instructions that make up a request are *nonprocedural.* That is, you can enter instructions in any order, with the following exceptions:

- Header instructions must come before any other instructions.
- The END DEFINITION instruction must be the last instruction in the request.

Although you can enter instructions in any order, at run time TDMS executes them in a predetermined order. For instance, output mappings are executed before input mappings. For a full listing of the order in which TDMS executes instructions, see Chapter 7, Instruction Execution Order, in the *VAX TDMS Reference Manual.*

The following sections describe the basic set of request instructions.

1.2.1 Using the FORM IS Instruction

Usually the first instruction you type in a request is the FORM IS instruction. This instruction identifies the form or forms you refer to in later instructions. You name a form using the format:

FORM IS form-path-name;

The form-path-name must be a legal CDD path name. You can select either the given, the full, or the relative path name. If the form is in your default CDD directory, you can use the given name in the FORM IS instruction. For example:

```
FORM IS EMPLOYEE_ADD_FORM;
```

You can also use a WITH NAME modifier with the FORM IS instruction. The WITH NAME modifier lets you assign a unique name to replace the CDD path name for the form. You must use this modifier if two forms, or a form and a record, in your request have the same given name. You can also use this modifier if you wish to use a short name throughout the request instead of the long CDD path name.

See Chapter 2, Using the Request Definition Utility (RDU) for more explanation of CDD path names.

If you have FORM IS instructions that refer to forms that are never displayed or used in a request, then you may receive error messages at run time and when you try to build a request library.

1.2.2 Using the RECORD IS Instruction

You must also name the CDD record definitions you will use later in mapping instructions within the request. You name a record using the following instruction:

RECORD IS record-path-name;

Like the form-path-name, the record-path-name you use can be the given, the full, or the relative path name. If a record definition is in your default CDD directory, you can use the given name with the RECORD IS instruction:

```
RECORD IS EMPLOYEE_RECORD;
```

You may have more than one RECORD IS instruction in a request. To identify several different records that the request uses, specify several RECORD IS instructions or use the single RECORDS ARE instruction and a list of record names.

You can also use a WITH NAME modifier with the RECORD IS instruction. The WITH NAME modifier lets you assign a unique name to replace the CDD path name for the record. You must use this modifier if two records, or a form and a record, in your request have the same given name. You can also use this modifier if you wish to use a short name throughout the request instead of the long CDD path name.

1.2.3 Using the DESCRIPTION Instruction and Comment Text

You can use the DESCRIPTION instruction anywhere in a request where you want to include descriptive text. The text you enter following the keyword DESCRIPTION and the slash and asterisk symbols (/*) is stored with the request in the CDD. You end the descriptive text with the asterisk and slash symbols (*/) and a semicolon. For example:

```
DESCRIPTION /*This is an example
   of a simple request.*/;
```

You can also include comments. To put a comment in your request, you put an exclamation point (!) at the beginning of each line. The comment can be placed on a separate line or after an instruction. (An example of a comment with an exclamation point is shown before the OUTPUT TO instruction in Figure 1-1.)

1.2.4 Using the CLEAR SCREEN Instruction

You can use the CLEAR SCREEN instruction to clear the terminal screen before you display a form on that screen. It ensures that there is nothing on the screen before you display a form. For example:

```
CLEAR SCREEN;
```

1.2.5 Displaying the Form

You identify which form a request can use by using the FORM IS instruction. In addition, you must also identify how a request is to use a particular form. For example, you can specify that TDMS:

- Display a form as defined in the form definition with the background text, the field default characteristics, and the field default contents that are assigned when a form is defined (the DISPLAY FORM instruction)
- Display a form with the background text and field contents from the last request call (the USE FORM instruction)

1.2.5.1 Using the DISPLAY FORM Instruction -- If the form definition contains default values for the form fields, these values are displayed when you specify the DISPLAY FORM instruction. However, if your request also contains output mappings, these output mappings override the form defaults. TDMS displays the data mapped for output rather than the form definition defaults. For example:

```
DISPLAY FORM EMPLOYEE_ADD_FORM;
```

Note that you specify the given name of the form (or the name specified in the WITH NAME clause to the FORM IS instruction) in the DISPLAY FORM instruction; you cannot use the full or the relative path name.

1.2.5.2 Using the USE FORM Instruction -- The USE FORM instruction displays a form as it looked (with the same background text and field contents) when the previous request call ended. If the form was not used in the previous request call, TDMS displays the form with its form definition defaults. For example:

```
USE FORM EMPLOYEE_CHANGE_FORM;
```

To see how you can use the DISPLAY FORM and the USE FORM instructions in requests, examine the following example of a TDMS application. The application program calls two requests, one with a DISPLAY FORM instruction (Request A) and one with a USE FORM instruction (Request B).

When the application program calls Request A, TDMS executes the DISPLAY FORM instruction and displays the form EMPLOYEE_INFO_FORM with all the form definition defaults. Assuming the form definition assigns a default value of 000000 to the BADGE field, this value is displayed when TDMS displays the form. The operator enters data in the form fields BADGE, NAME, and SEX.

Request A

```
RECORD IS EMPLOYEE_RECORD;
FORM IS EMPLOYEE_INFO_FORM;

     DISPLAY FORM EMPLOYEE_INFO_FORM;
     INPUT BADGE TO BADGE,
           NAME  TO NAME,
           SEX   TO SEX;
END DEFINITION;
```

← First program call (to Request A)

EMPLOYEE_INFO_FORM

```
BADGE: 000000
NAME: _________
SEX: _
```

← TDMS displays form and default value

(continued on next page)

EMPLOYEE_INFO_FORM

```
BADGE: 954678
NAME: Hartwood
SEX: F
```

← Operator enters data

When the program calls the next request in the application, Request B, TDMS executes the USE FORM instruction and displays EMPLOYEE_INFO_FORM again. However, in this instance, the form fields you see, BADGE, NAME, and SEX, contain all the data entered in those fields during the call to Request A.

Request B

```
RECORD IS EMPLOYEE_RECORD;
FORM IS EMPLOYEE_INFO_FORM;

      USE FORM EMPLOYEE_INFO_FORM;
      WAIT;
END DEFINITION;
```

← Second program call (to Request B)

EMPLOYEE_INFO_FORM

```
BADGE: 954678
NAME: Hartwood
SEX: F
```

← TDMS displays data that was collected in the call to Request A

Note that a USE FORM instruction displays the form context from a previous request call only if the form was used in the immediately previous request call.

1.2.6 Moving Data to and from the Form

The instructions that cause TDMS to move data between the form and the record are called **mapping instructions.** The basic mapping instructions in TDMS are:

- The INPUT TO instruction
- The OUTPUT TO instruction
- The RETURN TO instruction

Using these mapping instructions, you can create requests that handle most of the forms input/output (I/O) for TDMS applications. Chapter 3, Mapping Between Form Fields and Record Fields, describes these instructions in detail. The following sections introduce the most important mapping instructions, INPUT TO and OUTPUT TO.

1.2.6.1 Using the INPUT TO Instruction -- If you want TDMS to collect data from a form field and return it to a record, you must use the INPUT TO or INPUT %ALL instruction. The general format of the INPUT TO instruction is:

INPUT form-field TO record field;

For example:

```
INPUT EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER;
```

The INPUT TO instruction causes TDMS to place the cursor in the form field named EMPLOYEE_NUMBER and return the value the operator enters in that field to the record field named EMPLOYEE_NUMBER.

The INPUT TO instruction in Figure 1-2 shows how you can use the input instruction to map data between the record definition and form definition. (Note that, in this example, the names of the form fields are the same as the background text displayed next to each form field. Note also that the EMPLOYEE_NUMBER field is defined as a Display Only field, which means the operator cannot enter data.)

The request shows that you can list a series of input phrases (LAST_NAME TO LAST_NAME, FIRST_NAME TO FIRST_NAME, and so on) after the keyword INPUT. Each input phrase is separated by a comma, and the last phrase in the series ends with a semicolon. Note also that you must specify a receiving record field for each form field.

EMPLOYEE_ADD_FORM

```
                 Employee Basic Information
                            A D D

        EMPLOYEE_NUMBER: 9999999

        NAME:
         +------------------------------------------+
         |    FIRST_NAME: AAAAAAAAAA                 |
         |MIDDLE_INITIAL: A                          |
         |     LAST_NAME: AAAAAAAAAAAAAAAAAAAA       |
         +------------------------------------------+
             ADDRESS:
                  STREET: XXXXXXXXXXXXXXXXXXXX
                    CITY: CCCCCCCCCCCCCCC
                   STATE: AA
                ZIP_CODE: CCCCC

         SEX: A           BIRTH_DATE: 99-AAA-99
```

EMPLOYEE_RECORD Definition

```
DEFINE RECORD EMPLOYEE_RECORD.
EMPLOYEE_RECORD STRUCTURE.
  EMPLOYEE_NUMBER     DATATYPE   UNSIGNED LONGWORD.
  EMPLOYEE_NAME STRUCTURE.
    LAST_NAME         DATATYPE   TEXT    20.
    FIRST_NAME        DATATYPE   TEXT    10.
    MIDDLE_INITIAL    DATATYPE   TEXT     1.
  END EMPLOYEE_NAME STRUCTURE.
  EMPLOYEE_ADDRESS STRUCTURE.
    STREET            DATATYPE   TEXT    20.
    CITY              DATATYPE   TEXT    15.
    STATE             DATATYPE   TEXT     2.
    ZIP_CODE          DATATYPE   TEXT     5.
  END  EMPLOYEE_ADDRESS STRUCTURE.
  SEX                 DATATYPE   TEXT     1.
  BIRTH_DATE          DATATYPE   DATE.
END EMPLOYEE_RECORD STRUCTURE.
END EMPLOYEE_RECORD.
```

EMPLOYEE_SAMPLE_REQUEST

```
FORM IS          EMPLOYEE_ADD_FORM;
RECORD IS        EMPLOYEE_RECORD;

CLEAR SCREEN;
DISPLAY FORM     EMPLOYEE_ADD_FORM;

  INPUT   LAST_NAME        TO LAST_NAME,
          FIRST_NAME       TO FIRST_NAME,
          MIDDLE_INITIAL   TO MIDDLE_INITIAL,
          STREET           TO STREET,
```

Figure 1-2: Mapping Between a Record Definition and a Form Definition

```
            CITY            TO CITY,
            STATE           TO STATE,
            ZIP_CODE        TO ZIP_CODE,
            SEX             TO SEX,
            BIRTH_DATE      TO BIRTH_DATE;

  END DEFINITION;
```

Figure 1-2: Mapping Between a Record Definition and a Form Definition (Cont.)

You can also specify %ALL with the INPUT instruction. The INPUT %ALL instruction causes TDMS to collect data from all the fields on the form that have identically named record fields.

In Figure 1-2, the request EMPLOYEE_SAMPLE_REQUEST can be rewritten as follows using the INPUT %ALL instruction since all the form fields have matching record fields with identical names:

```
FORM IS          EMPLOYEE_ADD_FORM;
RECORD IS        EMPLOYEE_RECORD;

CLEAR SCREEN;
DISPLAY FORM     EMPLOYEE_ADD_FORM;

INPUT %ALL;

END DEFINITION;
```

The INPUT %ALL instruction maps form fields to record fields with identical names (LAST_NAME, FIRST_NAME, MIDDLE_INITIAL, STREET, CITY, STATE, ZIP_CODE, SEX, and BIRTH_DATE).

RDU does not create an input mapping for EMPLOYEE_NUMBER even though you specify %ALL. EMPLOYEE_NUMBER is defined on the form as a Display Only field, which means TDMS does not allow the operator to enter data in the field and RDU does not create an input mapping for that field.

1.2.6.2 Using the OUTPUT TO Instruction -- You can describe what data you want TDMS to move from the program record and display on the form using the OUTPUT TO instruction. The general format of the OUTPUT TO instruction is:

OUTPUT record-field TO form-field;

You can list a series of output phrases after the keyword OUTPUT. Each phrase must be separated by a comma. For example:

```
OUTPUT FIRST_NAME TO FORM_FIELD_1,
       LAST_NAME  TO FORM_FIELD_2;
```

If you want to output data to all the fields on a form (that have record fields with identical names), you can use %ALL. If you use %ALL, TDMS displays data to all those form fields that have identically named record fields. For example in the following request, the OUTPUT %ALL instruction outputs data in all the form fields listed after the DESCRIPTION instruction:

```
                    .
                    .
                    .
OUTPUT %ALL;

  DESCRIPTION /*
       EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER,
       LAST_NAME       TO LAST_NAME,
       FIRST_NAME      TO FIRST_NAME,
       MIDDLE_INITIAL  TO MIDDLE_INITIAL,
       STREET          TO STREET,
       CITY            TO CITY,
       STATE           TO STATE,
       ZIP_CODE        TO ZIP_CODE,
       SEX             TO SEX,
       BIRTH_DATE      TO BIRTH_DATE   */;
                    .
                    .
                    .
```

1.2.6.3 Using the WAIT Instruction -- If a request performs output mappings only, and does not include an INPUT TO instruction, you must use the WAIT instruction to ensure that the operator sees any messages or data you display on a form before TDMS clears the screen. The form remains on the screen until the operator acknowledges it by pressing the RETURN key or a program request key (PRK). See Chapter 11, Program Request Keys, for more information. The format of the WAIT instruction is:

WAIT;

1.2.7 Using Video Field Instructions

If you want to highlight certain fields on the form, you can use the video field instructions to specify what video attributes the field has. For example, the UNDERLINE FIELD instruction lets you specify that a field be underlined when the form is displayed. Or you can use %ALL to change the video attributes of all the fields on the form. The general format for this instruction is:

UNDERLINE FIELD form-field;

TDMS executes the UNDERLINE FIELD instruction when it executes the output mappings. UNDERLINE FIELD is one of five request instructions that affects the video attributes of form fields. The others are:

- BOLD FIELD
- BLINK FIELD
- REVERSE FIELD
- RESET FIELD

See Chapter 3, Request and Request Library Instructions in the *VAX TDMS Reference Manual*, for more information on using video field instructions.

1.2.8 Using the END DEFINITION Instruction

To identify the end of the request, you use the END DEFINITION instruction. When RDU encounters an END DEFINITION instruction, it returns you to the RDU> prompt and checks that your mappings are valid according to TDMS mapping rules. It checks that the form and record fields have compatible data types, lengths, and so on. For more information on how RDU decides whether a mapping is valid, see Chapter 4, Making Sure Your Request Mappings Are Correct.

Note that you must put a semicolon after the END DEFINITION instruction.

1.3 Rules for Entering Request Instructions

When you are entering request instructions, you should keep the following syntax rules in mind:

- Header instructions must appear first in the request definition.
- All instructions must end with a semicolon (;).

  Note that if comment text begins with the DESCRIPTION keyword, then it is part of a DESCRIPTION instruction and must end with a semicolon (;).

  Comment text that follows an exclamation point (!) does not need to end with a semicolon.
- A single instruction can extend over many lines or several instructions can appear on one line as long as each instruction is separated by a semicolon (except the MESSAGE LINE IS instruction and all quoted strings). For example:

  ```
  CLEAR SCREEN; DISPLAY FORM EMPLOYEE_ADD_FORM;
  ```

- Certain instructions can have modifiers. Modifiers are statements usually beginning with the keyword WITH.
- A list of field names must be separated by commas.
- A series of INPUT, OUTPUT, or RETURN phrases must be separated by commas. A phrase is a portion of a full instruction, in this case the portion that names the form and record fields between which data is mapped.
- Two or more receiving field names in a mapping instruction must be enclosed in parentheses. For example:

```
INPUT BADGE TO (BADGE, ID);
```

- All requests must end with an END DEFINITION instruction followed by a semicolon.

Refer to the *VAX TDMS Reference Manual* for information on all the request and request library instructions.

Using the Request Definition Utility (RDU) 2

You use the Request Definition Utility (RDU) to create, modify, and delete requests and request library definitions. To invoke the Request Definition Utility from DCL level, enter the following command:

```
$ RUN SYS$SYSTEM:RDU
```

The system responds with:

```
RDU>
```

Another method you might find easier is to define RDU as a symbol. First, make sure your system manager has not already assigned a meaning to the symbol RDU by using the SHOW SYMBOL command at DCL level:

```
$ SHOW SYMBOL RDU
%DCL-W-UNDSYM, undefined symbol - check validity and spelling
```

If no meaning has been assigned to the symbol RDU, you can define it as a symbol for entering the Request Definition Utility. Using a text editor (such as EDT), edit your login command file to define RDU as a symbol. Include the following line in your login command file:

```
$ RDU :== $RDU
```

Once you execute your new login command file, you can invoke the Request Definition Utility by typing the symbol RDU:

```
$ @LOGIN.COM
$ RDU
RDU>
```

The system responds with the RDU> prompt, which indicates you are at the utility level. If the RDU> prompt does not appear on your screen, see your system manager.

Once you are in RDU, you can:

- Issue commands to create requests and request library definitions
- Issue commands to manipulate (modify, delete, copy, list, replace, and so on) requests and request library definitions

The following sections explain how to perform these actions and which RDU commands to use.

2.1 How RDU Uses the CDD

RDU connects directly with the CDD when you issue certain commands (CREATE, MODIFY, REPLACE, COPY, DELETE, LIST, or VALIDATE) in RDU. When you enter request text in RDU, the utility checks the records and forms you specify in that text against the information the CDD has about these records and forms. If the request text contains correct references and it is otherwise valid, RDU stores the request in a CDD directory.

2.1.1 Setting RDU to Your CDD Directory

By default, when you enter RDU, it automatically connects to the CDD directory CDD$TOP. For RDU to connect to the CDD directory of your choice, you must define the logical name CDD$DEFAULT to point to that CDD directory, or you must use the command SET DEFAULT within RDU.

2.1.1.1 Defining CDD$DEFAULT -- If you define the logical CDD$DEFAULT to point to a personal CDD directory, RDU automatically connects to that directory when you invoke the utility and issue the CREATE command (or the MODIFY, REPLACE, COPY, DELETE, LIST, or VALIDATE commands).

To define CDD$DEFAULT from DCL level, use the DEFINE command. For example:

```
$ DEFINE CDD$DEFAULT CDD$TOP.YOUR_DIRECTORY
```

Now when you invoke RDU and issue commands, RDU connects to the CDD directory CDD$TOP.YOUR_DIRECTORY. This logical definition remains until you log off your computer or until you use another DEFINE command to change the definition.

You can insert the DEFINE CDD$DEFAULT command in your login command file. Then, when you log in, CDD$DEFAULT is set to the CDD directory of your choice.

2.1.1.2 Using the SET DEFAULT Command

-- When you are in RDU and want to set the CDD directory location, you must use the RDU command SET DEFAULT. For example:

```
RDU> SET DEFAULT CDD$TOP.YOUR_DIRECTORY
```

RDU automatically stores the requests and request library definitions that you create in this directory. When you use the SET DEFAULT command in RDU, the dictionary setting remains only until you exit RDU or until you issue another SET DEFAULT command.

You can insert the SET DEFAULT command in an RDU startup command file. This file, RDUINI.COM, contains commands that you want RDU to execute each time you invoke the utility. See the @file-spec command in the *VAX TDMS Reference Manual* for more information.

2.1.2 Using the SHOW DEFAULT Command

When you are in RDU, you can check which CDD directory RDU is connected to by typing:

```
RDU> SHOW DEFAULT
```

RDU responds with:

```
Current CDD default path is '_CDD$TOP.YOUR_DIRECTORY'
```

2.1.3 Naming Requests and Specifying Forms and Records

Because you are creating requests and request library definitions and referring to forms and records from the CDD, the names you use must be legal CDD path names.

A path name is the CDD name that indicates where in the CDD the request, request library, record, or form definition is stored. A legal CDD path name can be:

- A full path name, that is, all the names in the path name starting with the topmost name, CDD$TOP and ending with the given name (CDD$TOP.YOUR_DIRECTORY.EMPLOYEE_REQUEST, for example)
- A relative path name, that is, a portion of the full path name (for example, YOUR_DIRECTORY.EMPLOYEE_REQUEST)
- A given name, that is, the last name in the full path name (for example, EMPLOYEE_REQUEST)

Whenever you use a given or relative name for a request (or any CDD object), RDU converts that name to its full CDD path name. It does this by checking the

CDD$DEFAULT or the current setting of your CDD directory. It adds that directory name to the given or relative name of your request or object and stores or accesses the request or object in that directory. For instance, if you set your directory or CDD default to CDD$TOP.YOUR_DIRECTORY and specify a request as EMPLOYEE_REQUEST, RDU adds that directory name to your request given name EMPLOYEE_REQUEST and stores the request in that directory:

```
CDD$TOP.YOUR_DIRECTORY.EMPLOYEE_REQUEST
```

(If you have not defined CDD$DEFAULT or set your CDD directory, RDU uses the system default and stores your request in the topmost directory, CDD$TOP.)

You can use a given name in:

- All the request commands that require you to specify a request or request library definition name (CREATE, REPLACE, MODIFY, LIST, and so on)
- The instructions that require you to specify a form, record, or request name (FORM IS, RECORD IS, DISPLAY FORM, USE FORM, REQUEST IS, and so on)

Note

The given name is the last name you type when specifying a path name. It is usually the same as the final name of the path name in the CDD. However, from DCL level, you can assign a logical name to the given name used in a request or request library definition. In that case, the final name of the object in the CDD can be different from the given name in your request. For instance, if you use the name EMPLOYEE_REQUEST in your request and you define, at DCL level, EMPLOYEE_REQUEST to be EMPLOYEE_REQUESTS.EMPLOYEE_ONE, the final name of the request in the CDD is EMPLOYEE_ONE.

You can use a full or relative path name in:

- The commands requiring you to specify a request or request library definition name (CREATE, REPLACE, MODIFY, LIST, and so on).
- The FORM IS, RECORD IS, and REQUEST IS instructions. You cannot use the full or relative path name in the USE FORM or DISPLAY FORM instructions or in references to records in other request instructions. With these instructions, you must use the given name only.

For all the rules concerning creating and using legal CDD path names, see the *VAX Common Data Dictionary Data Definition Language Reference Manual.*

2.2 Creating a Request

Once you assign a default CDD directory, you are ready to create requests. There are two ways to create a request:

- The **interactive method.** Once you enter RDU, you issue the CREATE REQUEST command and enter instructions. RDU prompts you for further input. When you finish entering the request text, RDU then stores it in the CDD.
- The **file method.** From either DCL or RDU level, you pass to RDU a VMS command file containing a CREATE REQUEST command and request text or a text file that contains just request text. RDU executes the command file and places the request in the CDD.

The first few times you create requests, you will probably want to use the interactive method. RDU prompts you for commands and returns error messages, when appropriate, as you enter the request text.

2.2.1 Using the Interactive Method

To create a request interactively, type the CREATE REQUEST command in response to the RDU> prompt and a request name in response to the prompt for a request name:

```
$ RDU
RDU> CREATE REQUEST
Request Name: EMPLOYEE_REQUEST
```

The name you assign the request will be the CDD given name. If you define CDD$DEFAULT to point to your personal CDD directory, RDU will store the request you are creating in that directory. For example, if you define CDD$DEFAULT to be CDD$TOP.YOUR_DIRECTORY, the preceding example would create a request with the CDD path name CDD$TOP.YOUR_DIRECTORY.EMPLOYEE_REQUEST.

If you get an error from the CREATE REQUEST command:

- CDD$DEFAULT does not point to an existing CDD directory or points to a directory you do not have privilege to create an object in. Change your default directory and try the CREATE REQUEST command again.
- A request named EMPLOYEE_REQUEST already exists in that directory. Either delete the existing request, or use the REPLACE command to replace the existing request.

If neither of these corrects the problem, see your system manager.

After you enter the command CREATE REQUEST and name the request, RDU displays the RDUDFN> prompt. This prompt indicates that RDU is ready to receive request instructions. Note, as the following example indicates, you can enter the request name following the CREATE REQUEST command rather than waiting for RDU to prompt you for a request name.

```
$ RDU
RDU> CREATE REQUEST EMPLOYEE_REQUEST
RDUDFN>
```

2.2.2 Using the File Method

To create a request using the file method, type the request text into a VMS file and pass the file to RDU using the CREATE REQUEST command:

```
RDU> CREATE REQUEST EMPLOYEE_REQUEST -
RDU>_ [YOURVMS.DIRECTORY]EMPSAMP.RDF
```

Another example is to pass the file to RDU at DCL level by typing the symbol RDU and the command:

```
$ RDU CREATE REQUEST EMPLOYEE_REQUEST -
$_ [YOURVMS.DIRECTORY]EMPSAMP.RDF
```

When you create a request using the file method, RDU looks for a default file type .RDF. Note that at DCL level, you must type the symbol RDU before you type the CREATE REQUEST command. (Note also that the hyphen at the end of the command line allows you to continue the command on the next line.)

The text file EMPSAMP.RDF in the directory YOURVMS.DIRECTORY contains the request text. The instructions in the file must follow the same format as those you enter interactively in RDU.

2.2.3 Using a Command File - DCL or RDU Level

You can also use a command file to create a request. First type the RDU command CREATE REQUEST and the associated request text into a command file, then pass the file to RDU using the @file-spec command:

```
RDU> @[YOURVMS.DIRECTORY]EMPSAMP.COM
```

You can also pass the file to RDU at DCL level by typing first the RDU symbol and then the @file-spec command:

```
$ RDU @[YOURVMS.DIRECTORY]EMPSAMP.COM
```

In the preceding example, the command file EMPSAMP.COM in the directory YOURVMS.DIRECTORY can contain the command CREATE REQUEST EMPLOYEE_REQUEST and the associated request text. The commands and instructions in the file must follow the same format as those you enter interactively in RDU. A single command file can contain any number of request commands and accompanying request instructions.

2.3 Correcting Errors

After you have entered all the request instructions, including the END DEFINITION instruction, RDU checks your request definition for syntax errors and invalid mappings. If there are any errors, RDU displays messages telling you where the errors are. Rather than typing the entire request definition over again, you can edit your last command by entering the EDIT command:

```
RDU> EDIT
```

The EDIT command calls the default system editor. (In this manual the EDIT command calls the VMS EDT editor. You can specify a text editor of your choice by defining the logical name RDU$EDIT. You learn how to do this in Chapter 5, Finding and Correcting Your Errors.) When you issue the EDIT command:

1. RDU calls your editor and displays the last command you entered (including any request instructions associated with that command)
2. You can correct your typing errors just as you would in a regular text file, using all of the editor's features
3. When you are done, exit from the editor by issuing the appropriate exit command
4. RDU executes the last command you entered and checks this corrected request text for further errors

If you corrected all the errors, RDU stores this request in the CDD. If you still have errors in your request text, RDU continues to display error messages and does not store the request in the CDD. Use the EDIT command again to correct those errors that RDU identifies.

2.4 Validating Requests

By default, RDU is in Validate mode when you invoke the utility. In Validate mode, when you create, modify, or replace a request, RDU checks that:

- The form names and record names you refer to are in the CDD.

- The form field names you specify in the request match the form field names in the CDD form definition.
- The record field names you specify in the request match the record field names in the CDD record definition(s).
- In a %ALL mapping, at least one form field has an identically named record field.
- The mappings you define in the request are valid mappings. See Chapter 4, Making Sure Your Request Mappings Are Correct, for more information.

If the request is valid, RDU stores the new request definition in the CDD. If you do not specify the /NOSTORE qualifier on the command, the request binary structure is stored in the CDD with the request. See the section in this chapter, Using the /[NO]STORE Qualifier, for more information on request binary structures.

Similarly, if you create, modify, or replace a request library definition in Validate mode, RDU checks that the requests or forms you specify are in the CDD. If they are, RDU stores the new request library definition in the CDD. See Chapter 12, Working with Request Libraries, for more information about request library definitions.

Figure 2-1 shows how using Validate/Novalidate mode can affect your design sequence.

2.4.1 Changing the RDU Validation Option

You can change the default Validate mode to Novalidate mode by using the SET NOVALIDATE command. Novalidate mode allows you to create the request before the form and/or record(s). If you use Novalidate mode, you save space in the CDD; however, building a request library that refers to that request will subsequently take more time because the BUILD LIBRARY command will have to validate the request.

Once RDU is in Novalidate mode, you can get validation by:

- Exiting RDU and reentering the utility. This resets RDU to Validate mode.
- Issuing the SET VALIDATE command. This resets RDU to Validate mode.
- Issuing the VALIDATE REQUEST command. This leaves RDU in Novalidate mode but validates the specific request.
- Issuing the VALIDATE LIBRARY command. This leaves RDU in Novalidate mode but validates the specific request library definition.

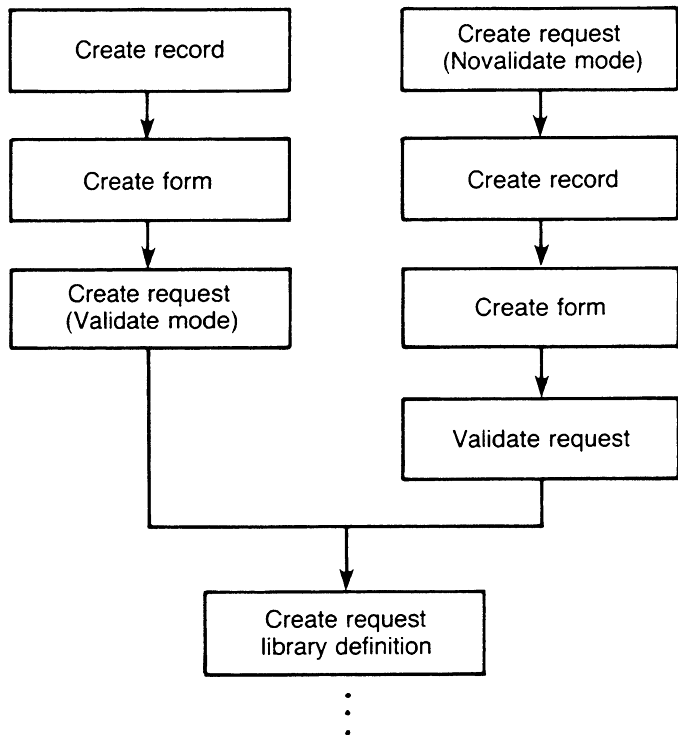


Figure 2-1: Suggested TDMS Design Sequence: Effects of Validation

2.4.2 Using the /[NO]STORE Qualifier

When RDU is in Validate mode, it also defaults to Store mode. That is, whenever you create, modify, replace, or validate a request, RDU stores a binary representation of the request in the CDD. Similarly, whenever you validate a request library definition, RDU stores the request binary structure for each request contained in the library. (Note that you do not see the request binary structure if you list the contents of the CDD directory. The request binary structure is simply an internal representation of the validated request and its mappings.)

If you validate a request library definition when the request binary structure is already stored in the CDD, RDU follows the same validation sequence that it follows for a request. See the section entitled How RDU Validates a Request for more information.

If you use the /STORE qualifier when RDU is in Novalidate mode, RDU signals an error and does not create, modify, replace, or validate a request.

If you use the /NOSTORE qualifier when RDU is in Validate mode, RDU does not store any request binary structures in the CDD. The size of the CDD decreases, but the time needed to build a request library file increases.

You should use the /NOSTORE qualifier only when RDU is in Novalidate mode or CDD space is limited. Unless CDD space is extremely important at your site, you should accept the TDMS default Validate mode and the resulting Store mode default.

2.4.3 How RDU Validates a Request

Validation of a request proceeds in three phases:

1. If the request was validated previously, RDU checks that any related forms or records have not changed. If the request has not changed, it is still valid.
2. If the request has not been validated previously, or if any related forms or records have changed, RDU validates the request.
3. RDU stores the request binary structure in the CDD. This third phase occurs only if:
   - The second phase was necessary
   - The validation process completed successfully
   - RDU is in Store mode

2.5 Copying Requests in the CDD

When you want to use a request, record, or form definition that already exists and is stored in another CDD directory, you have two choices:

- Leave the request in that directory and refer to it with a full CDD path name
- Copy the request, form, or record into your personal CDD directory and refer to it with the given name

To copy a request, use the COPY REQUEST command:

```
RDU> COPY REQUEST -
RDU>_ CDD$TOP.TDMS$SAMPLES.EMPLOYEE.EMPLOYEE_ADD_REQUEST -
RDU>_ EMPLOYEE_ADD_REQUEST
```

RDU creates a new request that is an exact copy of the first request you specify. The original request is left unchanged.

2.6 Modifying a Request

When you copy a request from another directory, it is very possible the request refers to form and record definitions stored in the original directory by their given names. To make sure the request can find these forms and records, you must modify the request to point to correct form and record path names.

To change an existing request, you use the MODIFY REQUEST command. The MODIFY REQUEST command extracts the requests from the CDD and calls the editor of your choice so you can edit the request in RDU. For example:

```
RDU> MODIFY REQUEST EMPLOYEE_ADD_REQUEST
```

You can now change the FORM IS and RECORD IS instructions to point to the forms and records that remain in the original directory by changing the given names to the full path names. After making your changes, type the appropriate command to exit your VMS editor and return to the RDU> prompt.

If you make no typing or spelling errors, RDU stores the modified request and its associated request binary structure in the CDD. If you make typing or spelling errors, RDU displays error messages and does not store the modified request. Instead, it asks whether you want to edit the request again. For example:

```
Do you want to re-try the MODIFY (y or n)?
```

If you enter a Y, RDU returns you to the editor and places the incorrect request text in the buffer for you to edit.

RDU continues to prompt you to retry the modify operation until the request text is correct. If you want to exit without making corrections, enter N. RDU discards all the request text and returns you to the RDU> prompt. In this case, the original request remains unchanged.

2.7 Listing a Request

To see a request that is stored in the CDD, use the LIST REQUEST command. The format is:

LIST REQUEST request-path-name;

For example, to list the EMPLOYEE_ADD_REQUEST request, type:

```
RDU> LIST REQUEST EMPLOYEE_ADD_REQUEST

Request          EMPLOYEE_ADD_REQUEST    1-JUL-1986 13:44:37   VAX RDU V1.6      Page 1
Source listing                           1-JUL-1986 13:44:37   SYS$INPUT:[].COM; (1)

   FORM IS        EMPLOYEE_ADD_FORM;
   RECORD IS        EMPLOYEE_RECORD;

   CLEAR SCREEN;
   DISPLAY FORM          EMPLOYEE_ADD_FORM;

      OUTPUT  EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER;
        !Displays the employee EMPLOYEE_NUMBER.  You do not
        !allow the operator to enter data in this
        !field: do not map the EMPLOYEE_NUMBER field for input.

      INPUT  %ALL;
        DESCRIPTION
         /* Inputs the following fields
         LAST_NAME     TO LAST_NAME,
         FIRST_NAME    TO FIRST_NAME,
            .
            .
            . */;

   END DEFINITION;
```

2.8 Deleting a Request

To delete a request from the CDD from within RDU, use the DELETE REQUEST comand. For example:

```
RDU> DELETE REQUEST YOUR_REQUEST
```

You can also delete a request from within the CDD Dictionary Management Utility (DMU), using the DMU DELETE command. These DELETE commands will remove the request from the CDD; they do not affect any VMS file in which a copy of the request might be stored. Before you delete a request, you might want to do one of the following:

- Enter RDU and copy the request to another name, such as YOUR_REQUEST_OLD. Copying the request within the CDD has the disadvantage that the size of the CDD remains the same.
- Enter DMU and use the EXTRACT command to copy the request from the CDD to a VMS file specification. Copying the request to a VMS file has the disadvantage of removing the request from the layers of security that might be present in the CDD. The size of the CDD, however, decreases.
- Print the request.

2.9 Exiting RDU

To exit from an RDU session and return to DCL command level, enter the EXIT command:

```
RDU> EXIT
$
```

You can also press CTRL/Z to exit from the utility.

Mapping Between Form Fields and Record Fields 3

The requests in Figures 1-1 and 1-2 contain mappings between form and record fields, using the INPUT TO and OUTPUT TO instructions. TDMS has a total of three mapping instructions:

- INPUT TO
- OUTPUT TO
- RETURN TO

With these instructions you use two different types of syntax:

- **%ALL syntax**, in which TDMS maps all form fields to and from record fields with the same name
- **Explicit syntax**, in which you specify the field names of the form and record fields you are mapping

In this chapter, you see examples of requests that map data between form and record fields using both types of syntax. In addition, this chapter describes:

- How the TDMS mapping instructions work
- How to specify fields in a mapping instruction
- When to use the %ALL mapping syntax
- When to use the explicit mapping syntax
- Using the %ALL and explicit syntax in the same request
- Mapping from a form to a group record field

3.1 How the TDMS Mapping Instructions Work

There are three mapping instructions:

- The INPUT TO instruction maps data from a form field to one or more record fields. You can use the INPUT TO instruction to let the operator enter data into a form field and return that data to the program in the record field when the request completes.

  If a form field is mapped for input but the operator does not enter data in that field, TDMS returns one of the following to the record field:

  - Data output to the form field by the current request
  - Data in the form field from the immediately previous request call (as a result of the USE FORM instruction)
  - Data associated with the form field by a form definition default (if no other data is in the field)

- The OUTPUT TO instruction maps data to one or more form fields. You can use the OUTPUT TO instruction to display data on the form when the request is invoked. The OUTPUT TO instruction uses either a quoted string or record field as a source of the mapping.
- The RETURN TO instruction maps data to one or more record fields. The RETURN TO instruction is similar to the INPUT TO instruction with the following exceptions:

  - The RETURN TO instruction uses either a form field or a quoted string as a source of the mapping.
  - The RETURN TO instruction does not open the field for input from the operator if the source of the mapping is a form field. Instead, it uses the current contents of that field.

  The RETURN TO instruction is very useful for conditionally returning data when the request completes or when the operator presses a predefined program request key (PRK).

The examples in this chapter use the INPUT TO and OUTPUT TO instructions to explain the general rules of field mapping.

3.2 How to Specify Fields in a Mapping Instruction

There are two types of mapping syntax:

- %ALL syntax
- Explicit syntax

When you use the %ALL syntax, you specify %ALL in place of the actual field names. For example:

```
OUTPUT %ALL;
```

When you use the explicit mapping syntax, you name the fields you want to map explicitly in the mapping instruction. The form and record field names you use in the mapping instructions are the field names as they appear in the record and form definitions. For example:

```
OUTPUT DATE_FIELD TO DATE_FIELD;
```

RDU uses the actual field names to create the mapping between the form and the record. By default, before RDU creates a mapping, it checks that:

- In a %ALL mapping, each form field matches one and only one record field with the same name
- In an explicit mapping:
  - The form field names you specify exist in the form definition that the request uses
  - The record field names you specify exist in the record definition(s) that the request uses
  - The record field name is unique in the record definitions

RDU also checks that the **structures** of form fields and record fields (group, indexed, simple, and so on) are compatible and that the **data types** (TEXT, NUMERIC, and so on) and the lengths of form and record fields are compatible.

Chapter 4, Making Sure Your Request Mappings Are Correct, discusses more about the form and record field structures and data types between which you can map. For now, however, as you examine the mapping examples in this chapter, notice that the form fields in the illustrations have **picture strings**. These picture strings describe which types of data an operator can enter in that form field and which kind of data a program can display in that field. For example, a form field with a picture string of AAA can accept data that is alphabetic and three characters long.

The record fields in the illustrations also describe what type of data they can contain by specifying a data type (TEXT, NUMERIC, UNSIGNED LONGWORD, and so on) next to the record field name. This data type must be compatible with the form field picture string before RDU will create the mapping.

3.3 When to Use the %ALL Syntax

The examples in this chapter show that you can use %ALL when you want to:

- Map to or from all the fields on a form (that have matching record fields).
- Reduce the number of mapping instructions you need to write.
- Increase the independence of requests from record and form definitions. (For example, you can add form and record fields to form and record definitions. The keyword %ALL maps these new fields without requiring any change to the request instructions.)

The examples in the following sections illustrate how to use the %ALL syntax to make your requests easy to create.

You can also use %ALL to map between form and record arrays. Examples and an explanation of how to map between form and record array fields are in Chapter 7, Mapping Between Form Arrays and Record Arrays.

3.3.1 Using %ALL to Map an Entire Form

With the %ALL syntax, you can display data in all the fields of a form and then collect data from all the fields on the form using only two instructions, OUTPUT %ALL and INPUT %ALL.

When you specify the OUTPUT %ALL and INPUT %ALL mapping instructions, TDMS:

- Displays data in all the form fields that have identically named matching record fields
- Collects data from all the form fields and returns it to identically named record fields

Note

For the purposes of explanation, the form field names used in the request are shown in the form definitions as background text next to the field picture strings.

EMPL_ADD_FORM

```
                Employee Basic Information
                          A D D

             EMPL_NUMBER: 9999999

           NAME:
          FIRST: AAAAAAAAAA
        INITIAL: A
           LAST: AAAAAAAAAAAAAAAAAAAA

        ADDRESS:
             STREET: XXXXXXXXXXXXXXXXXXXX
               CITY: AAAAAAAAAAAAAAA
              STATE: AA
           ZIP_CODE: CCCCC

     SEX: A             BIRTH_DATE: 99-AAA-99
```

EMPL_ADD_RECORD

```
DEFINE RECORD EMPL_ADD_RECORD.
EMPL_RECORD STRUCTURE.
 EMPL_NUMBER DATATYPE  UNSIGNED LONGWORD.
 NAME       STRUCTURE.
    LAST       DATATYPE    TEXT    20.
    FIRST      DATATYPE    TEXT    10.
    INITIAL    DATATYPE    TEXT     1.
 END NAME   STRUCTURE.
 ADDRESS    STRUCTURE.
    STREET     DATATYPE    TEXT    20.
    CITY       DATATYPE    TEXT    15.
    STATE      DATATYPE    TEXT     2.
    ZIP_CODE   DATATYPE    TEXT     5.
 END  ADDRESS        STRUCTURE.
    SEX        DATATYPE    TEXT     1.
    BIRTH_DATE DATATYPE    DATE.
 END EMPL_RECORD STRUCTURE.
END   EMPL_ADD_RECORD.
```

EMPL_ADD_REQUEST

```
FORM IS          EMPL_ADD_FORM;
RECORD IS        EMPL_ADD_RECORD;

CLEAR SCREEN;
DISPLAY FORM     EMPL_ADD_FORM;

        INPUT %ALL;
        OUTPUT %ALL;

END DEFINITION;
```

When you use the %ALL syntax for input and output (and return) mappings, RDU:

- Checks first to identify the form field names on the active form definition (EMPL_NUMBER, FIRST, INITIAL, LAST, STREET, CITY, STATE, ZIP_CODE, SEX, BIRTH_DATE)
- Then searches the record definitions for record field names identical to the form field names

After RDU determines that the form field names have identically named record fields, it checks that the fields have compatible field structures, data types, and lengths. See Chapter 4, Making Sure Your Request Mappings Are Correct, for more information.

If RDU finds an error in a mapping instruction implied by the %ALL syntax, it does not create a mapping for that single incorrect implied mapping. RDU does, however, create a request in the CDD that contains all the remaining correct mappings implied by the %ALL syntax, if the rest of the request is also valid. The entire request fails only if all the implied mappings are incorrect.

At run time, TDMS executes only the correct mapping instructions.

Note that the form field EMPL_NUMBER is a Display Only field. When you specify an INPUT %ALL mapping for a form that includes a Display Only field, RDU does not create an input mapping for that field. Instead, if the /LOG qualifier is specified, RDU displays an information message indicating that it cannot create an input mapping for a Display Only form field.

3.3.2 Using %ALL to Map Between a Form and a Larger Record

You can use the %ALL syntax to map between a form and a record that contains more fields than the form. RDU creates mappings to and from only those form fields that have identically named record fields.

For instance, in the following example, RDU creates successful output and input mappings between the form and record fields NUMBER, FIRST, MID_INIT, and LAST.

Because RDU uses the form as the key for %ALL mappings, including both input and output mappings, it ignores the remaining unmapped record fields: UNIVERSITY, DEGREE, START_DATE, STOP_DATE.

EMPLOYEE_HEADER_FORM

```
Employee Header Form

NUMBER: 9999999

NAME:
  FIRST:     AAAAAAAAAA
  MID_INIT:  A
  LAST:      AAAAAAAAAAAAAAAAAAAA
```

EDUCATION_INFO_REC

```
DEFINE RECORD EDUCATION_INFO_REC.
EDUCATION_REC STRUCTURE.
 NUMBER DATATYPE   UNSIGNED LONGWORD.
 EDUC_NAME STRUCTURE.
    LAST   DATATYPE     TEXT 20.
    FIRST  DATATYPE     TEXT 10.
    MID_INIT DATATYPE   TEXT 1.
 END EDUC_NAME STRUCTURE.
 UNIVERSITY     DATATYPE TEXT   18.
 DEGREE         DATATYPE TEXT   15.
 START_DATE     DATATYPE DATE.
 STOP_DATE      DATATYPE DATE.
END   EDUCATION_REC STRUCTURE.
END   EDUCATION_INFO_REC.
```

ALL_LARGER_REC_REQUEST

```
FORM IS        EMPLOYEE_HEADER_FORM;
RECORD IS      EDUCATION_INFO_REC;

CLEAR SCREEN;
DISPLAY FORM   EMPLOYEE_HEADER_FORM;

        OUTPUT %ALL;
        INPUT %ALL;

END DEFINITION;
```

3.3.3 Using %ALL to Map Between a Form and a Smaller Record

You can use the %ALL syntax to map between the fields on a form and a record definition with fewer fields than the form. For instance, in the following example,

the form has eight fields but the record has only four fields. RDU generates successful mappings to and from form fields that have identically named matching record fields: EDUC_NUMBER, FIRST, MID_INIT, and LAST.

At run time, TDMS executes the four correct mapping instructions between the form and record fields EDUC_NUMBER, FIRST, MID_INIT, and LAST.

Because RDU cannot identically match the names of form fields UNIVERSITY, DEGREE, START_DATE, and STOP_DATE, it does not create these individual mapping instructions. (If you specify the /LOG qualifier with the CREATE, MODIFY, REPLACE, VALIDATE or BUILD commands, RDU generates information level messages indicating the fields that it mapped. See Chapter 5, Finding and Correcting Your Errors, for more information on the types of error messages RDU displays.)

EMPLOYEE_EDUCATION_FORM

```
                Employee Education Information
                             Add

        EDUC_NUMBER: 9999999

      NAME:
              FIRST: AAAAAAAAAA
           MID_INIT: A
               LAST: AAAAAAAAAAAAAAAAAAAA

         UNIVERSITY: CCCCCCCCCCCCCCCCCC
             DEGREE: CCCCCCCCCCCCCCC

         START_DATE: 99/AAA/99
          STOP_DATE: 99/AAA/99
```

EDUCATION_INFO_RECORD

```
DEFINE RECORD EDUCATION_INFO_RECORD.
EDUCATION_RECORD STRUCTURE.
 EDUC_NUMBER DATATYPE  UNSIGNED LONGWORD.
 EDUC_NAME STRUCTURE.
     LAST   DATATYPE     TEXT 20.
     FIRST  DATATYPE     TEXT 10.
     MID_INIT DATATYPE   TEXT 1.
  END EDUC_NAME STRUCTURE.
 END EDUCATION_RECORD STRUCTURE.
 END   EDUCATION_INFO_RECORD.
```

EMPL_EDUCATION_REQUEST

```
FORM IS        EMPLOYEE_EDUCATION_FORM;
RECORD IS      EDUCATION_INFO_RECORD;

CLEAR SCREEN;
```

```
DISPLAY FORM   EMPLOYEE_EDUCATION_FORM;

        OUTPUT %ALL;
        INPUT  %ALL;

END DEFINITION;
```

3.4 When to Use Explicit Mapping Syntax

You must use explicit mapping syntax when you want:

- To specify each mapping
- To map record fields and form fields that do not have identical names
- To have RDU reject a request unless it can create a mapping for every instruction in the request
- To map a single form or record field to several receiving form or record fields
- To map to or from several record fields that do not have unique field names within the record definitions used by a request

3.4.1 Explicitly Mapping Between a Form and a Record

Unlike a request using the %ALL syntax, if a single mapping is incorrect when explicitly mapping between form and record fields, the entire request fails. RDU generates error messages for the incorrect mappings and does not create the request. For example, in the following request, RDU cannot find matching record fields for the record and form fields you explicitly name in the input mapping (UNIVERSITY, DEGREE, START_DATE, and STOP_DATE). *This request fails.*

EMPLOYEE_EDUCATION_FORM

```
          Employee Education Information
                       Add

     EDUC_NUMBER: 9999999

    NAME:
          FIRST: AAAAAAAAAA
       MID_INIT: A
           LAST: AAAAAAAAAAAAAAAAAAAA

      UNIVERSITY: CCCCCCCCCCCCCCCCCC
          DEGREE: CCCCCCCCCCCCCCC

      START_DATE: 99/AAA/99
       STOP_DATE: 99/AAA/99
```

(continued on next page)

EDUCATION_INFO_RECORD

```
DEFINE RECORD EDUCATION_INFO_RECORD.
EDUCATION_RECORD STRUCTURE.
 EDUC_NUMBER DATATYPE  UNSIGNED LONGWORD.
 EDUC_NAME STRUCTURE.
     LAST   DATATYPE   TEXT 20.
     FIRST  DATATYPE   TEXT 10.
     MID_INIT DATATYPE   TEXT 1.
  END EDUC_NAME STRUCTURE.
 END EDUCATION_RECORD STRUCTURE.
 END   EDUCATION_INFO_RECORD.
```

EMPL_EDUC_REQUEST

```
FORM IS        EMPLOYEE_EDUCATION_FORM;
RECORD IS      EDUCATION_INFO_RECORD;

CLEAR SCREEN;
DISPLAY FORM    EMPLOYEE_EDUCATION_FORM;

! This request is not correct

        INPUT  EDUC_NUMBER TO EDUC_NUMBER,
               FIRST       TO FIRST,
               LAST        TO LAST,
               MID_INIT    TO MID_INIT,
               UNIVERSITY  TO UNIVERSITY,
               DEGREE      TO DEGREE,
               START_DATE  TO START_DATE,
               STOP_DATE   TO STOP_DATE;
        OUTPUT EDUC_NUMBER TO EDUC_NUMBER,
               FIRST       TO FIRST,
               LAST        TO LAST,
               MID_INIT    TO MID_INIT;

END DEFINITION;
```

3.4.2 Using Explicit Syntax to Map from One Field to Many Fields

You can create a request that explicitly inputs, outputs, or returns one field to many receiving fields. This is called a one-to-many mapping. For example, the following request demonstrates how to map a single form field to two record fields.

The INPUT TO instruction maps a single form field, NUMBER, to two record fields, EDUC_NUMBER and FAMILY_NUMBER. In this example, the two record fields are within two separate records, EDUCATION_RECORD and FAMILY_RECORD. RDU searches these two records for the specified record field names. You do not need to specify a record name for each field, as long as the record field names are unique. Note also that the items in the list of receiving record fields are separated by commas and enclosed in parentheses.

EMPLOYEE_HEADER_FORM

```
            Employee Basic Information
                       Add

            NUMBER: 9999999
```

EDUCATION_RECORD

```
DEFINE RECORD EDUCATION_RECORD.
EDUCATION_RECORD STRUCTURE.
  EDUC_NUMBER DATATYPE   UNSIGNED LONGWORD.
END EDUCATION_RECORD STRUCTURE.
END   EDUCATION_RECORD.
```

FAMILY_RECORD

```
DEFINE RECORD FAMILY_RECORD.
 FAMILY_RECORD STRUCTURE.
  FAMILY_NUMBER DATATYPE  UNSIGNED LONGWORD.
 END FAMILY_RECORD STRUCTURE.
END  FAMILY_RECORD.
```

EMPLOYEE_INFO_REQUEST

```
FORM IS         EMPLOYEE_HEADER_FORM;
RECORD IS       EDUCATION_RECORD;
RECORD IS       FAMILY_RECORD;

CLEAR SCREEN;
DISPLAY FORM    EMPLOYEE_HEADER_FORM;

 INPUT NUMBER TO (EDUC_NUMBER, FAMILY_NUMBER);
END DEFINITION;
```

The following example performs a similar one-to-many mapping. However, this example illustrates an output mapping that maps a single record field to two form fields on the same form. The OUTPUT TO instruction maps a single record field, EDUC_NUMBER, to two form fields, NUMBER and BADGE.

EMPLOYEE_HEADER_FORM

```
            Employee Basic Information
                       Add

            NUMBER: 9999999
             BADGE: 9999999
```

EDUCATION_FAM_RECORD

```
DEFINE RECORD EDUCATION_FAM_RECORD.
 EDUCATION_RECORD STRUCTURE.
  EDUC_NUMBER DATATYPE  UNSIGNED LONGWORD.
 END EDUCATION_RECORD STRUCTURE.
END   EDUCATION_FAM_RECORD.
```

EMPLOYEE_FAM_REQUEST

```
FORM IS         EMPLOYEE_HEADER_FORM;
RECORD IS       EDUCATION_FAM_RECORD;
CLEAR SCREEN;
DISPLAY FORM    EMPLOYEE_HEADER_FORM;
   OUTPUT  EDUC_NUMBER TO (NUMBER, BADGE);
END DEFINITION;
```

Note the following about one-to-many mappings:

- In an INPUT TO or RETURN TO mapping:
  - One form field can map to several record fields
  - The record fields can exist in the same record, or in different records
- In an OUTPUT TO mapping:
  - A single record field can map to two or more form fields.
  - The form fields must be on a single, active form. You cannot map to more than one form.

You cannot output, return, or input many fields to one receiving field. RDU issues an error message and does not create a request if the request contains a many-to-one mapping.

You cannot use %ALL in a one-to-many mapping. The %ALL syntax requires an identical, unambiguous match between a single form field name and a single record field name.

3.4.3 Making Explicit References Unique

In the previous sections, all the request examples show mappings to record fields that have unique names within the record definitions used by that request. However, you may need to map to one or more record definitions that contain fields that have the same names.

For example, the record definition used in this section, EMP_INFO_RECORD, contains two of each of the following fields: LAST, FIRST, and MID_INIT.

EMPLOYEE_HEADER_FORM

```
              Employee Header Form

              EMPLOYEE NO.: 9999999

              NAME:
                FIRST:     AAAAAAAAAA
                MID_INIT:  A
                LAST:      AAAAAAAAAAAAAAAAAAAA
```

EMP_INFO_RECORD

```
DEFINE RECORD EMP_INFO_RECORD.
EMP_FAM_RECORD STRUCTURE.
FAMILY_INFORMATION STRUCTURE.
 FAM_NUM DATATYPE  UNSIGNED LONGWORD.
 SPOUSE_INFO STRUCTURE.
   SPOUSE_NUM DATATYPE UNSIGNED LONGWORD.
   NAME STRUCTURE.
     LAST     DATATYPE   TEXT 20.
     FIRST    DATATYPE   TEXT 10.          <--- Fields with
     MID_INIT DATATYPE   TEXT 1.                duplicate names
   END NAME STRUCTURE.                          (LAST, FIRST,
  END SPOUSE_INFO STRUCTURE.                    MID_INIT)
```

(continued on next page)

```
  CHILD_INFO STRUCTURE.
    CHILD_NUM DATATYPE UNSIGNED LONGWORD.
    NAME STRUCTURE.
      LAST      DATATYPE   TEXT 20.
      FIRST     DATATYPE   TEXT 10.
      MID_INIT DATATYPE   TEXT 1.
    END NAME STRUCTURE.
  END CHILD_INFO STRUCTURE.
END FAMILY_INFORMATION STRUCTURE.
END EMP_FAM_RECORD STRUCTURE.
END EMP_INFO_RECORD.
```

← Fields with duplicate names (LAST, FIRST, MID_INIT)

When record field names are the same, you must use unique names to refer to these fields. If the references you make are not unique, RDU returns an error message stating that your mapping reference is ambiguous. An ambiguous reference is a reference to a record field name that matches more than one record field within the records used by that request. If references to these duplicate record field names are not unique, RDU cannot resolve the references.

For example, the following request contains ambiguous field references. It maps the form fields FIRST, MID_INIT, and LAST to the record fields FIRST, MID_INIT, LAST. Because each record field exists twice in the record EMP_INFO_RECORD, this request will fail. RDU cannot resolve the ambiguous reference to the two fields. It generates an error message and does not create this request in the CDD.

EMPLOYEE_INFO_REQUEST

```
FORM IS          EMPLOYEE_HEADER_FORM;
RECORD IS        EMP_INFO_RECORD;

CLEAR SCREEN;
DISPLAY FORM     EMPLOYEE_HEADER_FORM;

 !  This request is not correct.

INPUT
  LAST      TO LAST,
  FIRST     TO FIRST,
  MID_INIT TO MID_INIT;

OUTPUT
  LAST      TO LAST,
  FIRST     TO FIRST,
  MID_INIT TO MID_INIT;

END DEFINITION;
```

Usually, to refer to record fields with the same name uniquely, you can:

- Qualify the field names with as many preceding **group field** names as necessary to make the record field name unique.

A group field is a field that contains other fields. For instance, in the EMP_INFO_RECORD record definition, the fields EMP_FAM_RECORD, FAMILY_INFORMATION, SPOUSE_INFO, CHILD_INFO, SPOUSE_INFO.NAME, and CHILD_INFO.NAME are all group fields. You can use these group field names to make the record field references unique.

- Qualify the field names with the record definition name. The record definition name is the top-level group name.

The following examples show how you can avoid ambiguous references by using preceding group field names and the record definition name.

Note

Not all references can be made unique by using preceding group field or record names. For ways to create unique field references in all cases, see Chapter 6, Rules for Resolving Ambiguous Field References, in the *VAX TDMS Reference Manual*.

Because you cannot qualify the %ALL syntax with group field or record definition names, you cannot use the %ALL syntax to map to record fields that are not unique within the record definitions used by a request.

3.4.3.1 Using Group Field Names

-- The following request maps to the identically named fields in the EMP_INFO_RECORD.

EMPLOYEE_FAMILY_FORM

```
            Employee Family Information
                       Add

          FAM_NUM: 9999999

   SPOUSE NAME:
         SFIRST: AAAAAAAAAA
      SMID_INIT: A
          SLAST: AAAAAAAAAAAAAAAAAAAA

     SPOUSE_NUM: 9999999

   CHILD NAME:
         CFIRST: AAAAAAAAAA
      CMID_INIT: A
          CLAST: AAAAAAAAAAAAAAAAAAAA

      CHILD_NUM: 9999999
```

(continued on next page)

EMP_INFO_RECORD

```
DEFINE RECORD EMP_INFO_RECORD.
EMP_FAM_RECORD STRUCTURE.
FAMILY_INFORMATION STRUCTURE.
 FAM_NUM       DATATYPE   UNSIGNED LONGWORD.
 SPOUSE_INFO STRUCTURE.
   SPOUSE_NUM DATATYPE UNSIGNED LONGWORD.
   NAME STRUCTURE.
     LAST      DATATYPE   TEXT 20.
     FIRST     DATATYPE   TEXT 10.
     MID_INIT DATATYPE   TEXT  1.
   END NAME STRUCTURE.
  END SPOUSE_INFO STRUCTURE.

 CHILD_INFO STRUCTURE.
   CHILD_NUM DATATYPE  UNSIGNED LONGWORD.
   NAME STRUCTURE.
     LAST      DATATYPE   TEXT 20.
     FIRST     DATATYPE   TEXT 10.
     MID_INIT DATATYPE   TEXT  1.
   END NAME STRUCTURE.
 END CHILD_INFO STRUCTURE.
END FAMILY_INFORMATION STRUCTURE.
END EMP_FAM_RECORD STRUCTURE.
END EMP_INFO_RECORD.
```

EMPLOYEE_INFO_REQUEST

```
 FORM IS          EMPLOYEE_FAMILY_FORM;
 RECORD IS        EMP_INFO_RECORD;

 CLEAR SCREEN;
 DISPLAY FORM     EMPLOYEE_FAMILY_FORM;

INPUT
  FAM_NUM     TO FAM_NUM,
  SLAST       TO SPOUSE_INFO.LAST,
  SFIRST      TO SPOUSE_INFO.FIRST,
  SMID_INIT   TO SPOUSE_INFO.MID_INIT,
  SPOUSE_NUM  TO SPOUSE_NUM,
  CLAST       TO CHILD_INFO.LAST,
  CFIRST      TO CHILD_INFO.FIRST,
  CMID_INIT   TO CHILD_INFO.MID_INIT,
  CHILD_NUM   TO CHILD_NUM;

OUTPUT
  FAM_NUM               TO FAM_NUM,
  SPOUSE_INFO.LAST      TO SLAST,
  SPOUSE_INFO.FIRST     TO SFIRST,
  SPOUSE_INFO.MID_INIT  TO SMID_INIT,
  SPOUSE_NUM            TO SPOUSE_NUM,
  CHILD_INFO.LAST       TO CLAST,
  CHILD_INFO.FIRST      TO CFIRST,
  CHILD_INFO.MID_INIT   TO CMID_INIT,
  CHILD_NUM             TO CHILD_NUM;
 END DEFINITION;
```

In this example, both the names of the record fields to which you map and some of the preceding group field names are not unique. The record fields LAST, FIRST, MID_INIT and the group field NAME occur several times within this single record.

Note that you need use only those preceding group field names necessary to make your field references unique. For instance, you need not specify all the preceding group field names, as in the following reference:

```
OUTPUT SPOUSE_INFO.NAME.LAST TO SLAST;
```

You can say, instead:

```
OUTPUT SPOUSE_INFO.LAST TO SLAST;
```

The intervening group field name, NAME, is not necessary to make the field reference complete.

Note that you refer to record fields using the same syntax, regardless of whether the record fields are in the same or different records.

3.4.3.2 Using the Record Name -- Two separate record definitions may contain fields for which all the field names are identical. For instance, in the following request both records contain the group name NAME and the field names LAST, FIRST, MID_INIT. There is no preceding group name that is unique.

If you map to two such identical record definitions, you can generally create a unique reference by using the record definition name. The record definition name is either:

- The given name that is specified in the RECORD IS instruction.
- The name specified by the WITH NAME modifier in the RECORD IS instruction. If the WITH NAME modifier is used in the RECORD IS instruction and you use a record name in a mapping instruction, you *must* use this name.

The request must specify the record names in the mapping instructions. Note that the record names are separated from field names by a period, just as field names within a record are separated by a period.

Note also that by using the record name to identify the fields uniquely, you can eliminate the intervening group name in the mapping reference. You need not specify, for instance:

```
OUTPUT FAMILY_RECORD.NAME.LAST TO LAST;
```

You can say, instead:

```
OUTPUT FAMILY_RECORD.LAST TO LAST;
```

The intervening group name, NAME, is not necessary to make the record reference unique.

EMPLOYEE_HEADER_FORM

```
            Employee Header Information

            NUMBER: 9999999

            NAME:
              FIRST:    AAAAAAAAAA
              MID_INIT: A
              LAST:     AAAAAAAAAAAAAAAAAAAA
```

FAMILY_RECORD

```
DEFINE RECORD FAMILY_RECORD.
FAMILIES_RECORD STRUCTURE.
FAMILY_NUMBER DATATYPE  UNSIGNED LONGWORD.
NAME STRUCTURE.
   LAST       DATATYPE  TEXT 20.
   FIRST      DATATYPE  TEXT 10.
   MID_INIT   DATATYPE  TEXT  1.
END NAME STRUCTURE.
END FAMILIES_RECORD STRUCTURE.
END FAMILY_RECORD.
```

DEPENDENT_RECORD

```
DEFINE RECORD DEPENDENT_RECORD.
FAMILIES_RECORD STRUCTURE.
FAMILY_NUMBER DATATYPE  UNSIGNED LONGWORD.
NAME STRUCTURE.
   LAST       DATATYPE  TEXT 20.
   FIRST      DATATYPE  TEXT 10.
   MID_INIT   DATATYPE  TEXT  1.
END NAME STRUCTURE.
END FAMILIES_RECORD STRUCTURE.
END DEPENDENT_RECORD.
```

EMPLOYEE_INFO_REQUEST

```
FORM IS         EMPLOYEE_HEADER_FORM;
RECORD IS       DEPENDENT_RECORD;
RECORD IS       FAMILY_RECORD;

CLEAR SCREEN;
DISPLAY FORM    EMPLOYEE_HEADER_FORM;

INPUT
   NUMBER   TO  (DEPENDENT_RECORD.FAMILY_NUMBER,
                FAMILY_RECORD.FAMILY_NUMBER),
   LAST     TO  (DEPENDENT_RECORD.LAST,FAMILY_RECORD.LAST),
   FIRST    TO  (DEPENDENT_RECORD.FIRST,FAMILY_RECORD.FIRST),
   MID_INIT TO  (DEPENDENT_RECORD.MID_INIT,
                FAMILY_RECORD.MID_INIT);

OUTPUT
    DEPENDENT_RECORD.FAMILY_NUMBER TO   NUMBER,
    DEPENDENT_RECORD.LAST          TO   LAST,
    DEPENDENT_RECORD.FIRST         TO   FIRST,
    DEPENDENT_RECORD.MID_INIT      TO   MID_INIT;

END DEFINITION;
```

Note

Using a record name will not always let you create unique field references. For ways to create unique field references in all cases, see Chapter 6, Rules for Resolving Ambiguous Field References, in the *VAX TDMS Reference Manual*.

You cannot use the %ALL syntax to map between a single form field and a record field that is not unique. If you create such a mapping, RDU generates an information message and does not create that instruction with the request in the CDD.

3.5 Using the %ALL and Explicit Syntax in the Same Request

You can create requests that contain both explicit and %ALL mappings for the same fields. You may want to do this if you use the %ALL syntax to reduce the number of request instructions but know that RDU will not create mappings for all the form fields indicated by the %ALL syntax.

For instance, the following request uses the %ALL syntax for both input and output mappings. RDU attempts to create mappings for all the form fields when it executes the %ALL mapping instructions.

EMPL_ENGINEER_FORM

```
                 Employee Engineer Information

                              ADD


          NAME:
            FIRST:   AAAAAAAAAA
            INITIAL: A
            LAST:    AAAAAAAAAAAAAAAAAAAA


          PROJECT_LEADER: AAAAAAAAAAAAAAAAAAAA
          PROJECT_NO:     999999999
          ENGINEER_NO:    999999999
```

EMPL_MAIN_RECORD

```
DEFINE RECORD EMPL_MAIN_RECORD.
EMPL_RECORD STRUCTURE.
 NAME       STRUCTURE.
    LAST       DATATYPE    TEXT    20.
    FIRST      DATATYPE    TEXT    10.
    INITIAL    DATATYPE    TEXT     1.
 END NAME   STRUCTURE.
END EMPL_RECORD STRUCTURE.
END   EMPL_MAIN_RECORD.
```

EMPL_ENGINEER_RECORD

```
DEFINE RECORD EMPL_ENGINEER_RECORD.
EMPL_RECORD STRUCTURE.
 NAME       STRUCTURE.
    LAST       DATATYPE    TEXT    20.
    FIRST      DATATYPE    TEXT    10.
    INITIAL    DATATYPE    TEXT     1.
 END NAME   STRUCTURE.

 PROJECT_LEADER   DATATYPE TEXT    20.
 PROJECT_NO       DATATYPE NUMERIC  9.
 ENGINEER_NO      DATATYPE NUMERIC  9.

END EMPL_RECORD STRUCTURE.
END   EMPL_ENGINEER_RECORD.
```

EMPLOYEE_INFO_REQUEST

```
FORM IS          EMPL_ENGINEER_FORM;
RECORD IS        EMPL_MAIN_RECORD;
RECORD IS        EMPL_ENGINEER_RECORD;

CLEAR SCREEN;
```

```
DISPLAY FORM   EMPL_ENGINEER_FORM;

 INPUT
    LAST     TO  (EMPL_MAIN_RECORD.LAST,
                 EMPL_ENGINEER_RECORD.LAST),
    INITIAL TO  (EMPL_MAIN_RECORD.INITIAL,
                 EMPL_ENGINEER_RECORD.INITIAL),
    FIRST    TO  (EMPL_MAIN_RECORD.FIRST,
                 EMPL_ENGINEER_RECORD.FIRST);
 INPUT %ALL;

 OUTPUT
    EMPL_MAIN_RECORD.LAST     TO LAST,
    EMPL_MAIN_RECORD.INITIAL TO INITIAL,
    EMPL_MAIN_RECORD.FIRST   TO FIRST;
OUTPUT %ALL;

END DEFINITION;
```

RDU can create mappings for the form fields PROJECT_LEADER, PROJECT_NO, and ENGINEER_NO but not for the form fields FIRST, INITIAL, and LAST. Instead, if the /LOG qualifier is specified, RDU generates informational messages and does not create mappings for the fields FIRST, INITIAL, and LAST. These fields exist twice in the records used by the request. A %ALL mapping to or from them results in ambiguous field references that RDU cannot resolve. You must use the explicit syntax in order to map these fields.

Thus, though you specify two mappings for the fields FIRST, INITIAL, and LAST, RDU creates only one set of mappings for these fields. At run time, TDMS has only one set of the mapping instructions to execute.

3.6 Mapping from a Form to a Group Record Field

Typically, when you map between a form and a record, you are moving the data in a single record field to a single form field. Under some circumstances, however, you may want to move the data in all the fields within a particular group field to a single form field.

For example, the following request illustrates how you can move all the data in the record fields AREA, EXCHANGE, and LOCAL by mapping between the group record field TELEPHONE and the form field TELEPHONE.

EMPL_ADD_FORM

```
                    Employee Basic Information
                                Add

             EMPLOYEE_NUMBER: 9999999

             NAME:
                 FIRST: AAAAAAAAAA
                   MID: A
                  LAST: AAAAAAAAAAAAAAAAAAAA

             TELEPHONE: (999)999-9999

        SEX: A          BIRTH_DATE: 99-AAA-99

                  For help press PF2 key
```

EMPL_RECORD

```
DEFINE RECORD EMPL_RECORD.
EMPL_RECORD STRUCTURE.
   EMPLOYEE_NUMBER  DATATYPE  UNSIGNED LONGWORD.
   NAME        STRUCTURE.
     LAST             DATATYPE  TEXT    20.
     FIRST            DATATYPE  TEXT    10.
     MID              DATATYPE  TEXT     1.
    END NAME STRUCTURE.
    TELEPHONE STRUCTURE.
      AREA            DATATYPE  TEXT     3.
      EXCHANGE        DATATYPE  TEXT     3.
      LOCAL           DATATYPE  TEXT     4.
    END TELEPHONE STRUCTURE.
    SEX               DATATYPE  TEXT     1.
    BIRTH_DATE        DATATYPE  DATE.
 END EMPL_RECORD STRUCTURE.
 END EMPL_RECORD.
```

EMPL_ADD_REQUEST

```
FORM IS          EMPL_ADD_FORM;
RECORD IS        EMPL_RECORD;

CLEAR SCREEN;
DISPLAY FORM     EMPL_ADD_FORM;

           INPUT  TELEPHONE TO TELEPHONE;
           OUTPUT TELEPHONE TO TELEPHONE;

END DEFINITION;
```

Note that the input and output mapping instructions refer to a simple form field called TELEPHONE, which consists of a single piece of data that can be divided into three pieces of information: area code, exchange, and local number. You are mapping this form field to and from the group field called TELEPHONE.

To make sure that the correct data is moved between the form and record at run time in this example, you must:

- Check that for input mappings:

  - The form field picture has the same number of characters as the group record field
  - The form field is defined as a Must Fill field

  This allows the operator to enter only data that matches the record field length. The form field TELEPHONE has a picture string of ten 9's. Because this is a Must Fill field, the operator must enter the full ten digits at run time.

- Check that for output mappings, the record fields identically match the length of the form field.

  RDU checks the length of the form field against the combined length of the fields within the group record field. (The backslash and hyphen are not part of the character count in a form field.) For instance, the record fields AREA, EXCHANGE, and LOCAL must contain ten characters (collectively) to correspond to the form field TELEPHONE, which is ten characters long.

When you map a simple form field to or from a group field, RDU does not check the data type of the sending or receiving field either on input or output mappings. In addition, TDMS does not check the data type at run time. Therefore, you should be very careful when creating such mappings.

Note that you cannot map a group record field to several form fields. If you map a group record field to or from a form field, that form field must be a single field.

You can also use %ALL to map between a form field and a group field.

Making Sure Your Request Mappings Are Correct 4

Chapter 3 discussed how to specify input and mappings using the request mapping instructions. However, even if the syntax of the mapping instruction is correct, the fields you are mapping may be incompatible. If the fields are not compatible, the request will fail. Therefore, it is important to know when mappings will be successful.

This chapter describes:

- Rules for creating correct mappings
- How to tell if record and form field structures are compatible
- How to determine if record and form field data types are compatible
- How to list and read form and record definitions

4.1 Rules for Mapping

When you create, replace, modify, or validate a request, validate a request library, or build a request library file (when RDU is in Validate mode), you must observe the following rules:

- The form and record definitions you refer to in the FORM IS and RECORD IS instructions must exist in the CDD.
- If you use explicit mapping:
  - Each record field name you specify must match one and only one record field name in the record definitions that the request uses
  - Each form field name you specify must match one form field name in the active form definition (the form definition specified in the DISPLAY FORM or USE FORM instruction)

- If you use the %ALL syntax, at least one form field must have an identically named matching record field.
- References to record fields with identical names must be unique.
- The form field structures and record field structures must be compatible. (For instance, an indexed form field must be mapped to a record array; see Chapter 7, Mapping Between Form Arrays and Record Arrays, for information on arrays.)
- The form field and record fields between which you map must be compatible in structure and data type.

4.2 Making Sure Record and Form Field Structures Are Compatible

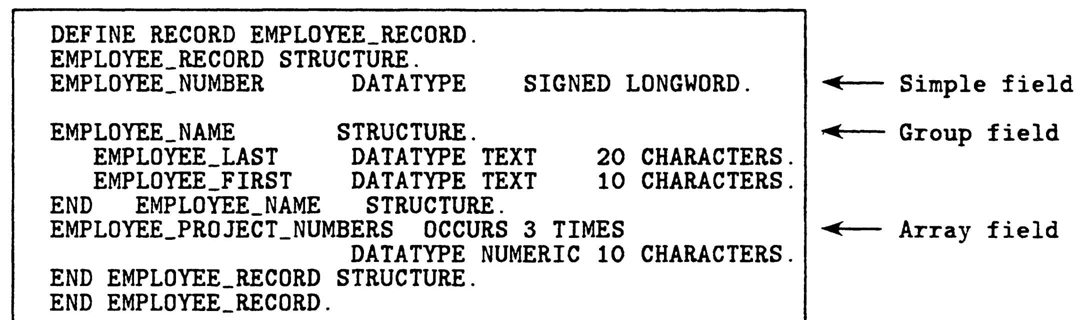
To create mappings between record and form fields, you must be sure that they have compatible field structures.

4.2.1 TDMS Record Field Structures

There are three distinct record field structures. The first two structures are simple and group fields (as described in Chapter 3). TDMS also supports record arrays. An array is a field that contains items (elements) that occur more than once. For more information on arrays, see Chapter 7, Mapping Between Form Arrays and Record Arrays.

Figure 4-1 shows the three record field structures: simple, group, and array.

EMPLOYEE_RECORD



```
DEFINE RECORD EMPLOYEE_RECORD.
EMPLOYEE_RECORD STRUCTURE.
EMPLOYEE_NUMBER       DATATYPE    SIGNED LONGWORD.

EMPLOYEE_NAME        STRUCTURE.
   EMPLOYEE_LAST      DATATYPE TEXT    20 CHARACTERS.
   EMPLOYEE_FIRST     DATATYPE TEXT    10 CHARACTERS.
END   EMPLOYEE_NAME    STRUCTURE.
EMPLOYEE_PROJECT_NUMBERS  OCCURS 3 TIMES
                      DATATYPE NUMERIC 10 CHARACTERS.
END EMPLOYEE_RECORD STRUCTURE.
END EMPLOYEE_RECORD.
```


Figure 4-1: Record Field Structures

4.2.2 TDMS Form Field Structures

There are two form field structures in TDMS:

- A simple field that contains a single piece of data
- A form array field (either indexed or scrolled) that, like a record array, contains items (elements) that occur more than once

Figure 4-2 shows the form field structures, simple and array.

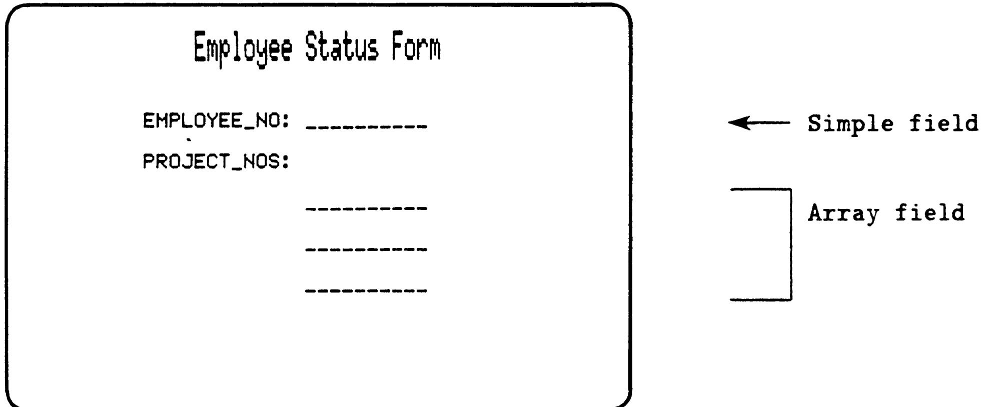


Figure 4-2: Form Field Structures

4.2.3 Compatible Form and Record Field Structures

You can map between the following form and record field structures (the numbers in brackets indicate specific elements in an array):

- A simple form field and a simple or a group record field:

```
INPUT EMPLOYEE_NO TO EMPLOYEE_NUMBER;
```

- A simple record or form field and a single element of a record or form array:

```
OUTPUT EMPLOYEE_NUMBER TO PROJECT_NOS[1];
```

- A simple record or form field and a set of elements in a record or form array:

```
OUTPUT EMPLOYEE_NUMBER TO PROJECT_NOS[1 TO 3];
```

- A form array and a record array:

```
OUTPUT EMPLOYEE_PROJECT_NUMBERS[1 TO 3] TO PROJECT_NOS[1 TO 3];
```

4.3 Making Sure Field Data Types and Length Are Compatible

To map between form and record fields, you must make sure that the data types of those fields are compatible. For instance, you can map a form field that is defined to accept alphabetic data to and from a record field with a text data type.

You must also be sure that the lengths of the fields are compatible. The length of a form or record field refers to the number of characters or digits that you can enter in that field. For example, to see what data types and field lengths are compatible, you must first understand the TDMS form field data types and the CDD record field data types presented in the following two sections.

4.3.1 TDMS Form Field Data Types and Lengths

When you create a form, you define the data type of each field by assigning a **picture string**. The picture string shows what type of data can be placed in the field and how long it can be. It combines the picture character with the field length.

A **picture character** (A,C,X, and so on) describes the type of data that can be entered in a form field (alphabetic, numeric, or date). TDMS assigns data types to these picture strings. For instance, if a field has a picture string of CCC, by default TDMS calls this a TEXT data type.

The number of times the picture character is repeated determines the length of a form field. For instance, a field with a picture string of AA is two characters long.

A field **size validator** (BYTE, UNSIGNED BYTE, and so on) determines both the type and size of data that can be entered in a form field.

When defining a form, you can assign size validators only for form fields that have a picture character of N or 9 (data type UNSIGNED NUMERIC or SIGNED NUMERIC). If assigned, the size validator (rather than the field picture) determines both the data type and the size of the form field.

For example, if you assign a size validator of UNSIGNED BYTE to a form field with a field picture of 999, TDMS calls this an UNSIGNED BYTE data type. The field can contain a value from 0 to 255.

For numeric fields, the length of the form field is also affected by the **scale factor.** The scale factor is a positive or negative integer that positions the decimal point in a numeric field. Scale factors can be assigned only to numeric fields with data types of UNSIGNED NUMERIC (NU) and SIGNED NUMERIC (NX).

For example, if a form field picture string is NNNN and the scale factor is 2, the total number of digits in the field is actually 6: there are 6 whole numbers on the left side of the decimal point. If, however, the picture string is NNNN and you assign a -2 scale factor, the total number of digits remains 4: there are 2 digits on each side of the decimal point. (A receiving or sending record field must have a compatible number of digits on both sides of the decimal point.) Table 4-1 shows the effect of the scale factor.

Table 4-1: Effect of Scale Factor on Form Field Data

Field Picture	Scale Factor	Data Entered on the Form	Value Sent to the Record
9999	0 (the default)	7234	7234
NNNN	2	7234	723400
99999	-2	49522	495.22

Table 4-2 shows which field picture characters can be used in each data type. Chapter 8, VAX TDMS Input and Output Mapping Tables in the *VAX TDMS Reference Manual*, contains a complete list of all the form and record field mappings RDU allows.

Table 4-2: TDMS Form Field Pictures and Form Field Data Types

Field Picture Type	Description	Data Type (Symbol)	Simplified Data Type Categories
C	Alphanumeric (A-Z, a-z, 0-9, space, and alphanumeric DEC Multinational characters)	TEXT (T)	Text
A	Alphabetic (A-Z, a-z, space, and alphabetic DEC Multinational characters)	TEXT (T)	Text

(continued on next page)

Table 4-2: TDMS Form Field Pictures and Form Field Data Types (Cont.)

Field Picture Type	Description	Data Type (Symbol)	Simplified Data Type Categories
X	Any displayable ASCII or DEC Multinational character	TEXT (T)	Text
Mixed	Combination C, A, X, N, 9	TEXT (T)	Text
9	Unsigned numeric (0-9)	UNSIGNED NUMERIC (NU)	Numeric
N	Signed numeric (plus sign, minus sign, 0-9, period)	SIGNED NUMERIC (NX)	Numeric
DATE and TIME	Date and time	DATE (ADT)	DATE

4.3.2 The Record Data Types to Which You Map

In the last section, you saw a list of the TDMS form field data types. Table 4-3 lists the CDDL record field data types that you can map to the TDMS form field data types. Only the subset of the CDDL record field data types that TDMS supports is listed in the table.

Record fields, like form fields, have a length. In the case of certain data types (for example, TEXT or SIGNED NUMERIC), the record definition explicitly specifies a field length in addition to the data type. Other data types (for example, WORD, BYTE, LONGWORD, F_FLOATING, H_FLOATING) have an implied length. Some record fields, like form fields, also have scale factors.

TDMS does not support the use of scale factors (implied decimal points) for binary integer or floating-point record field data types. Any scale factor appearing with the CDD record definitions of such fields is ignored when you build a request library file. If you want to use scaled record fields and have TDMS correctly handle decimal point alignment at mapping time, you must use one of the numeric string or packed-decimal record data types.

For the description of these CDDL record field data types and how record field length and scale factor are determined, see the *VAX Common Data Dictionary Data Definition Language Reference Manual.*

As with the TDMS form field data types, the CDD record field data types in Table 4-3 are categorized to help you understand, in general, the types of mappings RDU allows.

Table 4-3: Record Field Data Types

CDD Record Field Data Types that TDMS Supports	Simplified Record Field Data Types
TEXT (T) VARYING TEXT (VT)	Text
SIGNED BYTE (B) SIGNED WORD (W) SIGNED LONGWORD (L) SIGNED QUADWORD (Q) UNSIGNED BYTE (BU) UNSIGNED WORD (WU) UNSIGNED LONGWORD (LU) UNSIGNED NUMERIC (NU) LEFT SEPARATE NUMERIC (NL) RIGHT SEPARATE NUMERIC (NR) LEFT OVERPUNCHED NUMERIC (NLO) RIGHT OVERPUNCHED NUMERIC (NRO) SIGNED NUMERIC (NZ) PACKED NUMERIC (P)	Numeric
F_FLOATING (F) D_FLOATING (D) G_FLOATING (G) H_FLOATING (H)	Numeric Floating-point
DATE (ADT)	Date

4.4 Creating Mappings Between Compatible Data Types

Read this section for general information about creating input and output mappings between form field and record field data types. If you need detailed information, see Chapter 8, VAX TDMS Input and Output Mapping Tables, in the *VAX TDMS Reference Manual.*

In most cases, if field length (including scale factor), size, and signs are compatible:

- You *can*:
  - Input a Numeric form field to a Text record field
  - Output a Text record field to a Numeric form field without a size validator
  - Output a Numeric record field to a Text form field
- You *cannot* input a Text form field to a Numeric record field

For both Tables 4-4 and 4-5:

- Y indicates that mapping is permitted if field length (including scale factor), size, and sign are compatible
- N indicates that mapping is not permitted

The explanatory notes following Table 4-5 apply to both tables.

Table 4-4: Simplified Compatible Input Mappings (Form Fields to Record Fields)

Form Field Data Type	Record Field Data Type		
	Text	Numeric	Numeric Floating-Point
Text	Y	N	N
Numeric	Y	Y	Y

Table 4-5: Simplified Compatible Output Mappings (Record Fields to Form Fields)

Record Field Data Type	Form Field Data Type		
	Text	Numeric	Numeric with Size Validator
Text	Y	Y	N
Numeric	Y	Y	Y
Numeric Floating-Point	Y	N	N

Notes to Tables 4-4 and 4-5

- An N entry indicates that RDU will not let you map between the field data types.

  You cannot map between fields if the data type of the sending field, when converted by TDMS, is not compatible with the receiving field data type.

  Note that RDU may *allow output* mappings from Text to Numeric data types, but it will *not allow input* mappings from Text to Numeric data types. This is because the program can test the contents of a field before it is mapped for output and thus prevent run-time errors.

- A Y entry indicates that, generally, you can map between the indicated data types if the length (including scale factor), size, and sign conditions of the fields are compatible. (There are special tests for floating-point and group fields; these tests are described in more detail in Chapter 8 of the *VAX TDMS Reference Manual*, VAX TDMS Input and Output Mapping Tables.)

  Keep in mind that:

  - The field size or length is generally compatible if the size or length of the receiving field is equal to or greater than the field size or length of the sending field. RDU takes the scale factors of the receiving and sending fields into account when determining if field lengths are compatible.

  - The sign condition is compatible if both sending and receiving fields are signed or if both sending and receiving fields are unsigned.

- RDU allows some *output* mappings even if the length, size, or sign condition of the sending and receiving fields are incompatible. For example, RDU issues a warning message but *creates the mapping* in the following cases:

  - You output a record field of size UNSIGNED LONGWORD to a form field with a picture of seven 9s. The warning level message indicates possible overflow of the form field at run time. However, RDU creates this mapping because the program can take action to prevent a value being entered in the record field that would cause a run-time error.

  - You output a signed record field to an UNSIGNED NUMERIC form field. You can design the program so that a negative value never appears in a record field that is to be mapped to an UNSIGNED NUMERIC form field.

- RDU does not create *input* mappings if the field size, length, or sign conditions of the sending and receiving fields are incompatible. For example, RDU issues an error message and does not create the input mapping if:

  - You input a form field with a picture of seven 9s (an UNSIGNED NUMERIC data type) to a record field with the type WORD. Although the data types are compatible according to Table 4-5, the form field length is not compatible with the record field size. A field length of 9999999 could contain a number like 9876543; however, the maximum value that a WORD can contain is 32767. Therefore, RDU does not create a mapping.

  - You input a SIGNED NUMERIC form field to an UNSIGNED NUMERIC record field. Though the data types are compatible, the application program cannot stop the operator from entering a negative value that could be returned to the unsigned record field, thus causing a run-time error. Therefore, RDU does not create the mapping.

4.5 Form Definition Listings

To see if form field data types, structures, and names are correct, you must list the form definitions that the request uses. The following section describes how to list the form definition.

4.5.1 How to List the Form Definition

By using the LIST FORM command in FDU you can put the form definition into a file and then display it on your terminal or print it. To use the LIST FORM command, invoke FDU and enter the LIST FORM command using the /OUTPUT qualifier to create a VMS file of the listing information. You can then type (or print) the file specified in the LIST FORM command. For example:

```
$ FDU
FDU> LIST FORM/OUTPUT=MY_FORM.LIS  ADD_EMPLOYEE_FORM
FDU> EXIT
$ PRINT MY_FORM.LIS
```

When you have a listing of the form definition, you can examine it for the information described in the next section. For more information, see the LIST FORM command in the *VAX TDMS Reference Manual*.

4.5.2 What You Need to Know About Form Definitions

The information you need from a form listing includes:

- The CDD path name of the form you will use in an application
- The name of each form field you are mapping
- The structure of each form field (for example, simple, scrolled, or indexed)

You may also need to know:

- The field picture type of each field (numeric, alphabetic, date, alphanumeric, and so on)
- The length of each field
- The scale factor of each field
- The data type TDMS assigned each form field (TEXT, SIGNED NUMERIC, UNSIGNED NUMERIC, and so on)
- If the form has a Display Only field (you cannot use the field in an input mapping)
- If a field is a Response Required field (the operator must enter data if you map the field for input)
- If a field has a default value (a value TDMS returns from a form field if that field is mapped for input and the operator or the request did not enter data in it)

- If the field has special field validators (these validators check operator-entered data against specified ranges, lists of acceptable data, or other characteristics)

4.6 Record Definition Listings

To see if record field data types, structures, and names are correct, you access the record definitions associated with the request you are creating. You must list them from the CDD to get all the information you need to create mappings.

4.6.1 How to List the Record Definition

By using the LIST command in the Dictionary Management Utility (DMU), you can put the record definition into a file and then display it on your terminal or print it.

To use the LIST command, invoke DMU at DCL level on your system (if DMU is not defined as a symbol, you should define it first) and enter the LIST command using the /FULL and /LISTING qualifiers to create a VMS file of the listing information. You can then type (or print) the file specified in the LIST command. For example:

```
$ DMU :== $DMU
$ DMU
DMU> LIST/FULL/LISTING=MY_RECORD.LIS  ADD_EMPLOYEE_RECORD
DMU> EXIT
$ PRINT MY_RECORD.LIS
```

When you have a listing of the record definition, you can examine it for the information described in the next section. For more information, see the LIST command in the *VAX Common Data Dictionary Data Definition Language Reference Manual*.

4.6.2 What You Need to Know About Record Definitions

Examine the listing for the following information that you need to map data to record fields:

- The CDD path names of all record definitions your application uses
- The name of each record field
- The structure of each record field (simple field, group field, or array)

You may also want to know:

- The data type of each record field (NUMERIC, WORD, TEXT, DATE and so on)
- The length or size of each field in the record
- The scale factor of each field (if any)

For a full understanding of CDDL record definition syntax, refer to the *VAX Common Data Dictionary Data Definition Language Reference Manual*.

Finding and Correcting Your Errors 5

If, when you create a request, you misspell a word, or refer to a form or record incorrectly, RDU responds with error messages. There are many possible errors that RDU checks for, to make sure your request is valid. This chapter describes the following types of errors RDU checks for when you create, replace, or modify a request definition:

- Syntax errors -- Errors in spelling and punctuation.
- Semantic errors -- Errors in request logic. (Semantic errors include errors in references to CDD form and record definitions and errors in the mapping instructions between form fields and record fields.)

This chapter also describes semantic errors that RDU cannot check for you.

5.1 Syntax Errors Found by RDU

When RDU creates, modifies, or replaces a request or a request library definition, it checks for syntax errors. Syntax errors include punctuation and spelling mistakes.

The following are examples of syntax errors:

- Missing semicolon at the end of an instruction
- Misspelled instruction keyword
- Missing commas between elements in a list
- Missing END DEFINITION instruction followed by a semicolon at the end of a request

Whenever RDU encounters syntax errors, it displays error level messages. Error level messages always result in the failure of a request. RDU does not create, modify, or replace a request or a request library definition that contains syntax errors.

For example, the following requests illustrate the types of syntax errors you can make and how RDU responds:

```
RDU> CREATE REQUEST EMPLOYEE_TEST_REQUEST         You forgot
RDUDFN> FORM IS  EMPLOYEE_MENU_FORM        <---   a semicolon.
RDUDFN> RECORD IS EMPLOYEE_RECORD

  0002            RECORD IS EMPLOYEE_RECORD
                  1
.....................
%RDU-E-NOSEMICLN, Missing ';' at end of previous instruction
RDUDFN>


RDU> CREATE REQUEST EMPLOYEE_TEST_REQUEST_2
RDUDFN> FORM IS  EMPLOYEE_ADD_FORM;               You misspelled
RDUDFN> DISPALY FORM EMPLOYEE_ADD_FORM;    <---   an instruction.

  0002             DISPALY FORM EMPLOYEE_ADD_FORM;
                   1
.....................
%RDU-E-MISPELKWD, misspelled keyword 'DISPALY'; should be 'DISPLAY'
RDUDFN>


RDU> CREATE REQUEST EMPLOYEE_TEST_REQUEST_3
RDUDFN> FORM IS  EMPLOYEE_ADD_FORM;
RDUDFN> RECORD IS  EMPLOYEE_ADD_REC;
RDUDFN> DISPLAY FORM EMPLOYEE_ADD_FORM;           You forgot the
RDUDFN>  OUTPUT                                   comma between
RDUDFN>  EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER <---  list elements.
RDUDFN>  EMPLOYEE_FIRST  TO EMPLOYEE_FIRST;

  0006             EMPLOYEE_FIRST  TO EMPLOYEE_FIRST;
                   1
.....................
%RDU-E-SYNTAXERR, Found 'EMPLOYEE_FIRST' when expecting 'with'
RDUDFN>
```

Note that, in Interactive mode, RDU displays syntax error messages when it encounters the error. However, it does not cancel the CREATE REQUEST command. You can continue to enter request instructions and RDU checks for further errors.

When you finish entering the source text, RDU indicates that the CREATE REQUEST instruction has failed and returns you to the RDU> prompt. At this point, you can use the EDIT command to correct the source text and have RDU perform the CREATE REQUEST command again. See the section entitled Using the EDIT Command for more information.

5.2 Semantic Errors Found by RDU

When RDU is in Validate mode and you are creating, replacing, modifying, or validating a request or building a request library file, RDU checks for semantic errors. Semantic errors are mistakes that occur either in the internal logic of a request or in the external logic that the request establishes between any forms and records referred to by the request. The most common semantic errors you find when working with a request are mapping errors.

Be aware that RDU cannot check all semantic errors. For more information, read the section in this chapter entitled, Semantic Errors Not Found by RDU.

5.2.1 Mapping Errors

The following are examples of mapping errors for which RDU generates messages:

- The record and form definitions you refer to in a request do not exist in the CDD.
- The record fields you specify in a mapping instruction do not exist in the record definition.
- The form fields you specify in a mapping instruction do not exist in the active form definition.
- None of the form fields in a mapping implied by the %ALL syntax have an identically named record field.
- A record field that you specify (or that is implied by a %ALL mapping) is not unique within all the record definitions used by a request.
- The form and record fields you map do not have compatible structures, data types, lengths, or sizes.

If you refer to a record or form definition in the header section that does not exist in the CDD, RDU issues an error level message whether the request contains %ALL or explicit mappings.

5.2.1.1 %ALL Warning and Information Messages -- In the case of %ALL mappings, if RDU finds mapping errors other than a nonexisting form or record definition, only the single incorrect mapping instruction fails. (If the /LOG qualifier is specified with the command, RDU issues information and warning level messages to indicate that the incorrect mapping instruction was not created). RDU creates all other correct mapping instructions implied by the %ALL syntax. The entire request fails only if all the implied mappings are incorrect.

For example, in the following request, the record EMPLOYEE_RECORD does not contain a field named PROJECT to match the form field PROJECT. RDU does not create a mapping between the fields PROJECT and PROJECT. It does, however, create the request with the two other mapping instructions.

Note that because you used the /LOG qualifier, RDU issues an information level message indicating both the mapping that failed and the request RDU created.

```
RDU>      CREATE REQUEST  EMPLOYEE_MENU_REQUEST/LOG
RDUDFN>      FORM IS      EMPLOYEE_MENU_FORM;
RDUDFN>      RECORD IS    EMPLOYEE_RECORD;
RDUDFN>      DISPLAY FORM EMPLOYEE_MENU_FORM;

RDUDFN>          OUTPUT  %ALL;
RDUDFN>          DESCRIPTION
                    /* %ALL outputs
                       LAST     TO LAST,
                       MID_INIT TO MID_INIT,
                       PROJECT  TO PROJECT */; <--- Form field PROJECT
                                                    has no identically
                                                    named record field.
RDUDFN>    END DEFINITION;

                  Request at EMPLOYEE_MENU_REQUEST
  0001          FORM IS EMPLOYEE_MENU_FORM;
......................1
%RDU-I-LODFRMNAM, loading form
%RDU-I-BLDFRMNAM, building form
  0002          RECORD IS EMPLOYEE_RECORD;
..........................1
%RDU-I-LODRECNAM, loading record
              Request at EMPLOYEE_MENU_REQUEST;
...............1
%RDU-I-BLDREQNAM, building request
A 0004OUTPUT EMPLOYEE_RECORD.LAST TO LAST
A 0004          OUTPUT EMPLOYEE_RECORD.MID_INIT TO MID_INIT
A 0004          OUTPUT PROJECT TO PROJECT
.......................1
%RDU-I-NOSUCHFLD,  no such field
%RDU-I-NOMAPCRE, no mapping created
%RDU-S-REQCREATE, request EMPLOYEE_MENU_REQUEST created
RDU>
```

5.2.1.2 Explicit Mappings and Error and Warning Level Messages -- For explicit mappings, if RDU finds mapping errors, it issues either warning or error level messages.

For example, in the following request, RDU cannot create a mapping between the data types of the form and record fields. TDMS does not allow mapping between the data types UNSIGNED WORD and DATE. It issues an error level message whether the mapping is an input, output, or return mapping.

```
RDU> CREATE REQUEST     EMPLOYEE_INFO_REQUEST
RDUDFN>      FORM IS      EMPLOYEE_INFO_FORM;
RDUDFN>      RECORD IS    EMPL_RECORD;
RDUDFN>      DISPLAY FORM EMPLOYEE_INFO_FORM;
```

```
RDUDFN>    OUTPUT
RDUDFN>      EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER,
RDUDFN>      DAY_NUM         TO DAY_DATE;            <--- Data type error
RDUDFN>    END DEFINITION ;

  0006                    DAY_NUM TO DAY_DATE;
.........................1
%RDU-E-ILLDSTDAT, Unsupported data type in destination of mapping
%RDU-E-NOMAPCRE, no mappings created
%RDU-E-ERRDPAR, Error during instruction processing
%RDU-E-NOREQCRE, no request created
RDU>
```

In some cases, RDU allows you to create mappings between fields with data types or field lengths, sizes, or signed conditions that may cause run-time errors if certain data is mapped. In these cases, RDU issues warning level messages but allows you to create the mappings. In the following example, for instance, the request outputs data between the form field EMPLOYEE_BADGE with a field picture of seven 9's and a record field EMPLOYEE_BADGE with a data type of SIGNED LONGWORD. RDU issues the warning level messages for this single mapping instruction indicating that:

- If a negative number exists in the record field at run time and is mapped to the form field, you get a run-time error.
- If a number with a length of ten digits is in the record field at run time, the data will be truncated. (The longest length of data a LONGWORD record field can contain is ten digits and the form field can contain a figure only seven digits long.)

Note that RDU does, however, allow you to create this request.

```
RDU> CREATE REQUEST  EMPLOYEE_ACCT_REQUEST
RDUDFN>   FORM IS    EMPLOYEE_ACCT_FORM;
RDUDFN>   RECORD IS  EMPL_ACCT_RECORD;

RDUDFN>   DISPLAY FORM EMPLOYEE_ACCT_FORM;

RDUDFN>        OUTPUT
RDUDFN>        EMPLOYEE_BADGE TO EMPLOYEE_BADGE;     <--- Sign and length
RDUDFN>END DEFINITION;                                    warning

                       REQUEST AT EMPLOYEE_ACCT_REQUEST
  0005                output employee_badge to employee_badge;
...................1
%RDU-W-CNDNEGSRC, Mapping may produce conversion error if source is negative
%RDU-W-DSTLENGEQ, DESTINATION LENGTH must be greater than or equal to 10

RDU>
```

5.2.2 Other Semantic Errors Found by RDU

If you include a DISPLAY FORM or USE FORM instruction in your request, but forget to include a corresponding FORM IS instruction in your request header, RDU signals an error.

If you specify a WITH OFFSET modifier for a DISPLAY FORM or USE FORM instruction, RDU checks that the combination of the offset value and the form size does not exceed 23. Your limit is 23 lines per screen; the 24th line is reserved for the terminal message line. For example, if you specify an offset value of 10, and the form itself is 15 lines long, RDU signals an error.

If you specify a WITH NAME modifier for two or more forms, records, or for a combination of form(s) and record(s), the name must be unique. RDU signals an error if two or more WITH NAME modifiers use duplicate names.

Chapter 6, Using Conditional Instructions in Requests, describes how to set up conditional instructions within a request using a CONTROL FIELD IS instruction. If you misuse a DISPLAY FORM or USE FORM instruction within the context of a CONTROL FIELD IS instruction, RDU signals an error.

5.3 Semantic Errors Not Found by RDU

RDU cannot check all semantic errors. Sometimes you set up instructions that are legal within RDU but generate unexpected results. The following information offers you a starting point for finding errors in your request.

5.3.1 Order Execution Errors

The order in which request instructions occur is not necessarily the order in which they execute.

For example, you might try to default a form field first, then output information to that field:

```
DEFAULT FIELD EMPLOYEE_NAME;
        .
        .
        .
OUTPUT SMITH TO EMPLOYEE_NAME;
```

Only one of these instructions is performed but TDMS does not guarantee which one.

When you place two OUTPUT TO instructions for a single field in two different parts of the base request, you appear to lose one of the instructions. Suppose you say:

```
OUTPUT A TO B_FIELD;
        .
        .
        .
OUTPUT C TO B_FIELD;
```

Only one of these OUTPUT TO instructions is performed but TDMS does not guarantee which one.

5.3.2 Mapping Errors

RDU cannot identify the following common mapping errors:

- An error in the relationship of a RECORD IS instruction to the request invocation call in the application program:
  - The *number* of records in the request must match the number of records in the programming call that invokes the request
  - The *order* of records in the request must match the order of the records in the programming call that invokes the request
- A %ALL mapping error if one form field matches one record field. For example, you can refer to the wrong record in your request, and a record field in that record can accidentally match a form field. In this case, RDU will validate the %ALL mapping, yet your results will not meet your expectations.

  Whenever you use %ALL syntax, double check to be sure that you refer to the forms and records that you intended.
- In Chapter 7, Mapping Between Form Arrays and Record Arrays, you learn some restrictions about arrays. For now, be aware that TDMS restricts the number of elements that you can use in the source and destination of an array mapping.

5.3.3 Form-Related Errors

Form-related errors that RDU cannot identify include the following:

- TDMS supports a single active form at any given time. If you leave one form on the screen from the previous request and add another form, using a DISPLAY FORM WITH OFFSET instruction, both forms appear on your screen at once. However, only the current form is considered active:
  - If you press CTRL/W in order to refresh your screen, you will lose the inactive form.
  - Any input, output, or return mappings must direct information to or from the active form. You cannot perform any mappings to or from the inactive form.
  - You cannot apply any video attributes or DEFAULT FIELD or RESET FIELD instructions to the inactive form.

- A USE FORM instruction causes TDMS to display the active form from the immediately preceding request.

  The preceding request call must use that form in a USE FORM or DISPLAY FORM instruction. Otherwise, the USE FORM instruction defaults to a DISPLAY FORM instruction. The result is that the data entered in form fields seems to be lost.

5.4 Correcting Errors

As mentioned in Chapter 2, you can use the RDU EDIT and MODIFY commands to correct any errors you detect in your requests.

5.4.1 Using the EDIT Command

The EDIT command places the last command and any associated request text in the edit buffer and allows you to correct or change the RDU command or the text using the editor's features. You can use it in RDU after you enter an RDU command (such as CREATE REQUEST, REPLACE REQUEST, SET DEFAULT, and so on) and any text that may be associated with that command. When you exit the editor and create a file, RDU executes the commands in the file.

If you exit the editor without creating an output file, RDU returns you to the RDU> prompt. It does not keep a copy of your request text in the RDU buffer.

5.4.2 Using the MODIFY Command

The MODIFY command allows you to make changes to requests or request library definitions that are already stored in the CDD. RDU extracts the request or request library definition from the CDD and displays it on your screen for editing. When you exit the edit buffer, if you made no syntax or mapping errors, RDU stores the modified request (or request library definition). If RDU finds additional errors, it displays error messages and does not store the modified request. Instead, it prompts you to indicate whether you want to edit the request or request library definition again. RDU continues to prompt you until the request text is correct. If you exit before you make all corrections, RDU discards all the request text and returns you to the RDU> prompt.

5.4.3 Defining RDU$EDIT

When you issue the EDIT command or MODIFY command, RDU translates a series of logical definitions to invoke the VMS EDT editor and your startup file (EDTINI.EDT), if one exists.

The logical RDU$EDIT points to a system-defined logical TDMS$EDIT. TDMS$EDIT in turn points to a system-defined command procedure file (SYS$COMMON:[SYSEXE]TDMSEDIT.COM). The command file invokes the editor EDT and your startup file. (This chain of logical definitions is set up at installation time.)

You can change the RDU EDIT and MODIFY commands to invoke a personal command procedure that points to a particular editor. To do so, you must define the process logical RDU$EDIT to call a command procedure you create. That procedure, in turn, invokes the editor with the characteristics you want.

5.4.4 Using the SAVE Command

Another way you may wish to make corrections to requests or request library definitions is to use the SAVE command. The SAVE command lets you save the last command entered and any associated text. RDU saves the text in the VMS file you specify:

```
RDU> SAVE
Save to file      : EMPREQ
%RDU-I-SAVETOFIL, previous command SAVEd to file DBA2:[SMITH]EMPREQ.SAV;1
RDU>
```

This command creates a file (in this example EMPREQ.SAV) in your current default directory, containing the last command you entered and the request or request library definition text associated with that command. The default file type is .SAV.

If you issue the TYPE command at DCL level, you can see the request (or request library definition) text and RDU commands in the file EMPREQ.SAV.

For example:

```
$ TYPE EMPREQ.SAV

  CREATE REQUEST EMPLOYEE_ADD
        FORM IS  EMPLOYEE_MENU_FORM;
        RECORD IS EMPLOYEE_SELECT_RECORD;

        CLEAR SCREEN;
        DISPALY FORM EMPLOYEE_ADD_FORM;         <--- Error
        OUTPUT
            EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER  <--- Error
            EMPLOYEE_LAST TO EMPLOYEE_LAST;
       END DEFINITION;
```

You can use the SAVE command if you want to create a file containing your request text and then make corrections to your request text in that file. Note that the file contains none of the error messages that RDU generates.

When you have made the necessary corrections, you can resubmit this file to RDU:

```
RDU>  @EMPREQ.SAV
```

Note that the file you create contains the last command you entered. Be sure that the command is the one you want RDU to execute before passing the file back to RDU.

Using Conditional Instructions in Requests 6

A **conditional instruction** is a request instruction that TDMS executes only if certain conditions are true. A request containing one or more of these instructions is called a **conditional request.** Whether or not TDMS executes a conditional instruction depends on a run-time value called a **control value.** This chapter discusses the general concept of a conditional instruction.

6.1 Using Conditional Instructions

You may want to use conditional instructions to:

- Simplify maintaining and developing an application by placing all terminal I/O instructions within fewer requests
- Simplify the programmer's task by removing code associated with logical statements from the program and placing it in conditional instructions within a request

For example, you can use conditional instructions to:

- Collect menu selections and direct the resulting flow of an application from a single request
- Select and display appropriate error messages, using a single request when one of a series of predefined run-time errors occurs
- Collect data from a selected group of elements within a form array, using a single request (you learn about collecting selected form array elements in Chapter 9, Using an Array as a Control Value)
- Output, input, and return data conditionally

6.2 Conditional Requests

A conditional request contains one or more conditional instructions. It contains:

- A header section with one or more of the following instructions:

  - FORM IS
  - RECORD IS

  As in a simple request, you must enter header instructions before any other request instructions.

- A base section containing instructions that TDMS executes each time a program calls the request.

- One or more conditional instructions. Each conditional instruction contains:

  - The key phrase CONTROL FIELD IS to identify the kind of conditional instruction to be executed. It is *not* followed by a semicolon.
  - A control value that identifies the record field TDMS evaluates during request processing. The run-time control value determines which set of instructions TDMS executes.

    The control value must be data type TEXT.

  - One more more case values associated with the control field. At run time, TDMS matches the case values you specify with the values in the control value. The case value must be a quoted string or the keyword ANYMATCH or NOMATCH.
  - Optional match instructions associated with each case value. A match instruction is a request instruction to be executed when a case value matches the control value.

    Each match instruction is a request instruction in itself and must therefore be followed by a semicolon (;).

  - The end phrase END CONTROL FIELD to indicate that this conditional instruction is complete: there are no more case values to compare with the current control value.

    Because the ending phrase concludes the conditional instruction, it must be followed by a semicolon (;).

- An END DEFINITION instruction followed by a semicolon (;) to indicate the end of the request.

Figure 6-1 illustrates the structure of a conditional request.

```
FORM   IS form-name;
RECORD IS record-name;

 request-instruction;
           .
           .
           .

 CONTROL FIELD IS control-value

   case-value:

      match-instruction;
           .
           .
           .

  [case-value:

      match-instruction;]
           .
           .
           .

 END CONTROL FIELD;

END DEFINITION;
```

Figure 6-1: A Conditional Request

6.3 Using Conditional Requests

When a TDMS application program runs, it places values in the control value. These values are usually determined by the program when it evaluates either data returned from the operator in a previous request call or data from a database.

Because the program places values in the control value to determine I/O action, it is necessary for the programmer to know:

- What fields you declare as control values
- Which case values you declare
- Which action you define as a result of a control value evaluation

You can make sure the programmer has such information by using comment text in the requests you create.

(Though the value in the control value is usually placed there by the action of the program, you can design a request that maps values to the control value, either directly from the operator or from literals within the request.)

When an application program calls conditional requests, the following action occurs:

1. TDMS checks your control value
2. TDMS compares the control value with all the case values that you associated with that control value
3. If the control value matches any of the case values associated with that control value, TDMS executes the instructions following the matching case value

For example, a request might contain an instruction to collect information about an employee's spouse only if a control value indicates the employee is married, as in Figure 6-2. Figure 6-2 shows a request containing a single CONTROL FIELD IS instruction. This single conditional instruction contains two sets of request instructions. TDMS executes one of these sets of instructions depending on the value in the WK_MARITAL field when the program calls this request.

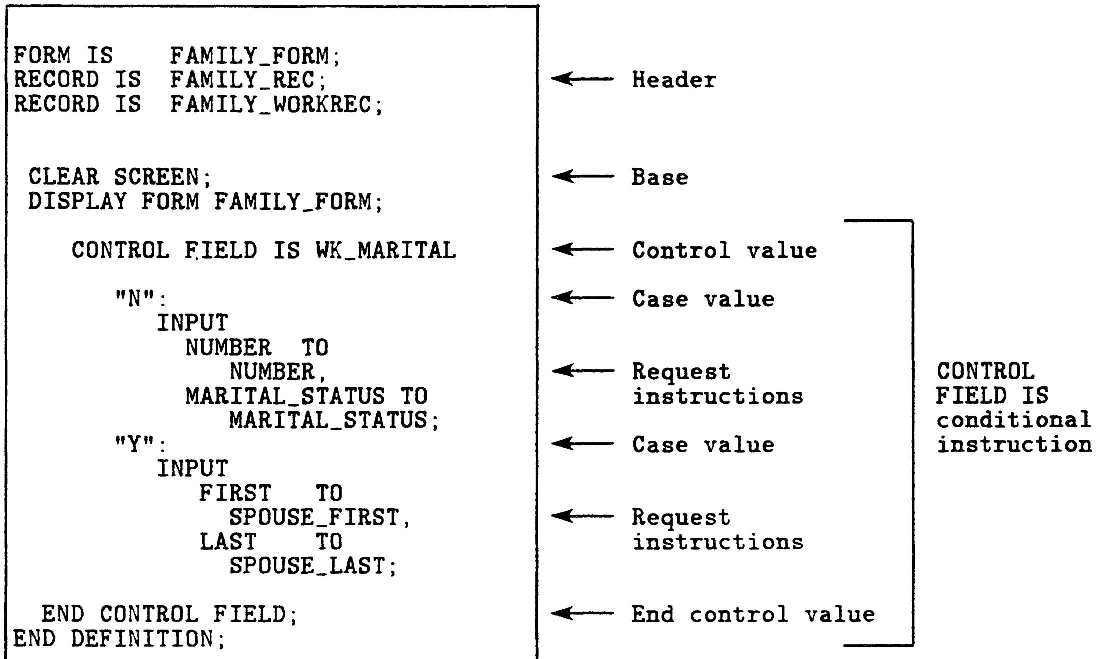


Figure 6-2: Request Containing a CONTROL FIELD IS Instruction

Notice that Figure 6-2 contains all the elements of a conditional request:

- A header section with FORM IS and RECORD IS instructions.
- A base section. The CLEAR SCREEN and DISPLAY FORM instructions are part of the base request.
- A conditional instruction, CONTROL FIELD IS, that contains:
  - A control value, WK_MARITAL. The application program places a "Y" or an "N" in this control value.
  - Two case values associated with that control value. Here the case values are two quoted strings, "N" and "Y".
  - Request instructions associated with each case value. In Figure 6-2 both case values are followed by INPUT TO instructions.
  - The end phrase END CONTROL FIELD.
- The END DEFINITION instruction.

6.4 How TDMS Executes a Conditional Instruction at Run Time

Figure 6-3 shows how the conditional request in Figure 6-2 works in a TDMS application.

At run time, the following sequence of events occurs:

1. The TDMS application program places an "N" (the default value) in the control value WK_MARITAL and calls the FAMILY_ADD request.
2. TDMS executes the FAMILY_ADD request and:
   - Displays the FAMILY_FORM
   - Collects an employee number and marital status
   - Returns the data to the program record
   - Terminates the request

3. The program evaluates the marital status. If the employee is married, the program places a "Y" in the control value WK_MARITAL, replacing the "N" that was already in that field. If the employee is not married, the program leaves the "N" in the control value.
4. The program calls the request a second time.
5. TDMS executes the FAMILY_ADD request and:

   - Displays the FAMILY_FORM
   - Collects spouse name if a "Y" is in the control value
   - Collects the employee number and marital status for a new employee if an "N" is in the control value
   - Terminates the request

Each time the application program calls the conditional request in Figure 6-2 TDMS executes only one set of instructions. The control value WK_MARITAL determines which set of instructions TDMS executes.

6.4.1 Specifying Control Values

It is often useful to define the control values in separate records that serve as workspaces. Although these workspace records are stored in the CDD, they are not used to contain data that is input or output to the terminal.

Workspace records are different from database records. Workspace records are used to collect information that the application program will use, then discard. Database records are used to collect permanent information. By using a separate workspace record to contain control values, you avoid the danger of mapping extraneous information to a database.

In your request, you name these workspace records in the RECORD IS instruction. When the application program or the request sets up control values, the database information is not affected.

For example, in Figure 6-3, the control value WK_MARITAL is a field in the workspace record FAMILY_WORKREC. The record FAMILY_REC contains the database information returned in record field MARITAL_STATUS.

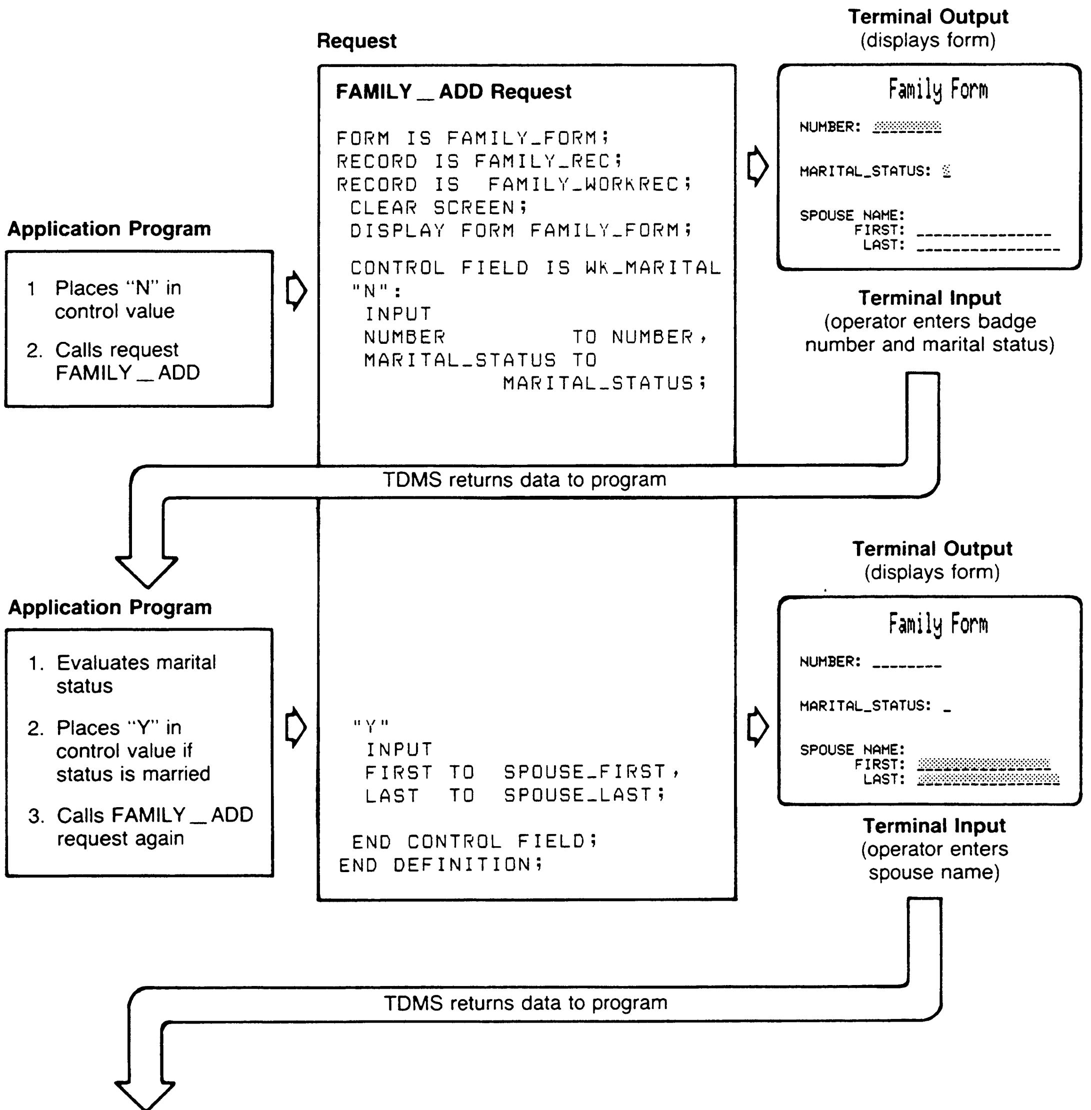


Figure 6-3: How a Conditional Request Works

6.4.1.1 Specifying More Than One Conditional Instruction

-- You can specify any number of CONTROL FIELD IS instructions in a single request. For every conditional instruction, you specify a single control value. However, you must be careful that the instructions under separate control values do not conflict.

For example, the following conditional request could result in conflicting mappings at run time. TDMS does not signal an error when such mappings occur.

The request contains two CONTROL FIELD IS instructions. The control values are EMPLOYEE_WAGE_CLASS and EMPLOYEE_JOB_CODE. If the program places an "S" in the first control value and an "M" in the second control value, TDMS tries to map two different literals, "SALARIED" and "CLASS 3", to the form field EMPLOYEE_STATUS.

```
CONTROL FIELD IS EMPLOYEE_WAGE_CLASS
    "H":  OUTPUT "HOURLY" TO EMPLOYEE_STATUS;
    "S":  OUTPUT "SALARIED" TO EMPLOYEE_STATUS;
END CONTROL FIELD;

CONTROL FIELD IS EMPLOYEE_JOB_CODE
    "J" :  OUTPUT "CLASS 1" TO EMPLOYEE_STATUS;
    "L" :  OUTPUT "CLASS 2" TO EMPLOYEE_STATUS;
    "M" :  OUTPUT "CLASS 3" TO EMPLOYEE_STATUS;
END CONTROL FIELD;
```

If the program does place an "S" in EMPLOYEE_WAGE_CLASS and an "M" in EMPLOYEE_JOB_CODE, TDMS executes only one of the mapping instructions (with no guarantee of which one). TDMS issues no warning message either at build time or at run time.

Note that case values are case insensitive. Any combination of upper and lower case letters in the control field produces a match. For example, either "H" or "h" will match the control value "H" in the previous example.

6.4.1.2 Using Nested CONTROL FIELD IS Instructions

-- You can *nest* layers of CONTROL FIELD IS instructions just as you nest IF statements in program code. For example, the following request illustrates the use of nested CONTROL FIELD IS instructions.

```
CONTROL FIELD IS JOB_CODE
    "H" :  OUTPUT "Hourly" TO WAGE_FIELD;
    "S" :  OUTPUT "Salaried" TO WAGE_FIELD;

           CONTROL FIELD IS BENEFIT_CODE
              "S"      :
                 OUTPUT "Participating in stock option"
                     TO BENEFITS_FIELD;
              NOMATCH :
                 OUTPUT "No benefits"
                     TO BENEFITS_FIELD;
           END CONTROL FIELD;
END CONTROL FIELD;
```

When a program calls this request, TDMS first evaluates the outer control value JOB_CODE, finds a matching case value, and executes the associated OUTPUT instructions.

TDMS executes the nested OUTPUT instructions only after it executes the outer OUTPUT instructions. The case value containing the nested control field instruction must match the control value in the outer CONTROL FIELD IS instruction.

For instance, in the preceding example, TDMS evaluates the conditional instruction CONTROL FIELD IS BENEFIT_CODE only if the value "S" is in the control value JOB_CODE. It executes the first of the nested mapping instructions to BENEFITS_FIELD only if the case value "S" appears in BENEFIT_CODE. Any other value causes TDMS to execute the second nested mapping instruction.

At run time, TDMS executes all INPUT instructions, both outer and nested, after it executes all OUTPUT instructions.

You can nest conditional instructions as many times as you want. You should be careful, however, to avoid mappings that may lead to conflicting run-time instructions when creating complex conditional instructions.

Note

Conflicting mapping instructions always result in only one mapping. If you have mapping instructions in a base request that conflict with mapping instructions in a conditional instruction, the conditional mappings override the mappings in the base request. If mapping instructions in a nested conditional instruction conflict with those in the outer conditional instruction, the nested mapping is executed.

6.4.2 Specifying Case Values

Each control value must have one or more associated case values. Any of the following is a legal case value for the CONTROL FIELD IS instruction:

- A quoted string
- NOMATCH
- ANYMATCH

6.4.2.1 Using the NOMATCH Case Value -- If you specify the NOMATCH case value and no other case values match the control value, TDMS executes the instructions associated with NOMATCH. In the following example, TDMS executes the instructions following the keyword NOMATCH only if the other case value, "FIRST", does not match the value in the control value, START_PROGRAM.

```
CONTROL FIELD IS START_PROGRAM

"FIRST":
        DISPLAY FORM EMPLOYEE_INITIAL_FORM;
        RETURN "     " TO START_PROGRAM;
        WAIT;

NOMATCH:
        DISPLAY FORM EMPLOYEE_MENU_FORM;
        INPUT       SELECTION TO SELECTION;
        INPUT EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER;

END CONTROL FIELD;
```

In the first call to this request, the value "FIRST" is in START_PROGRAM and the RETURN instruction returns an empty string to the control value, START_PROGRAM. This string clears the control value for the next call to the request.

In the next call, the control value no longer matches the case value "FIRST", so TDMS executes the NOMATCH instructions.

Note that if you conditionally reference forms and you are using the NOMATCH case value, make sure you have a DISPLAY FORM or USE FORM in the NOMATCH case value. Otherwise, RDU cannot determine which form is active.

If you are not conditionally referencing forms, put the DISPLAY FORM or USE FORM instructions in the base request.

6.4.2.2 Using the ANYMATCH Case Value -- You can also specify the ANYMATCH case value. If TDMS finds any matches between a control value and its associated case values, it executes not only those instructions but also the ANYMATCH instructions.

In the following example, TDMS executes the instructions following the keyword ANYMATCH if either of the other case values ("DUPLICATE" or "NO_RECORD") match the control value, SELECT_ERROR.

```
CONTROL FIELD IS SELECT_ERROR

"DUPLICATE":
     OUTPUT "EMPLOYEE RECORD ALREADY EXISTS"
                     TO  FIRST_MESSAGE_FIELD,
            "ENTER NEW NUMBER OR NEW SELECTION"
                     TO SECOND_MESSAGE_FIELD;

"NO_RECORD":
     OUTPUT "NO RECORD EXISTS WITH THAT NUMBER."
                     TO FIRST_MESSAGE_FIELD,
            "CHECK NUMBER AND TRY AGAIN."
                     TO SECOND_MESSAGE_FIELD;
```

```
ANYMATCH:
     OUTPUT EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER,
            SELECTION       TO SELECTION;
     INPUT  EMPLOYEE_NUMBER TO EMPLOYEE_NUMBER,
            SELECTION       TO SELECTION;

END CONTROL FIELD;
```

Note that the form fields EMPLOYEE_NUMBER and SELECTION are input and output each time an error message is displayed on the form. Each time, therefore, that the program detects a run-time error:

1. The application program places an appropriate value indicating the error in the control value
2. The program calls the request
3. TDMS executes the instructions if the control value matches one of the case values
4. The operator can enter a new employee number and a new menu selection if any one of the errors specified in the request occurs

6.4.2.3 Conditional Use of Forms -- It is best not to use the ANYMATCH case value when you want to conditionally reference forms in case values. Instead, repeat all mappings and include a DISPLAY FORM or USE FORM instruction within each case value.

However, if you must use ANYMATCH in this situation, be aware that a special situation exists when you try to conditionally reference forms in case values. RDU cannot know which case values will be executed at run time. When you have a mapping instruction in the ANYMATCH case value and DISPLAY FORM or USE FORM instructions in any other case value instructions, RDU cannot determine what form, (if any), will be active at run time.

In addition, the TDMS run-time system does not retain context about control field case values and associated instructions once the case value and instructions are executed. Therefore, when ANYMATCH instructions are executed at run time, TDMS does not know about any form referenced in any other case value instructions within the control field.

Keep the following points in mind when you conditionally reference forms in CONTROL FIELD IS instructions that use the ANYMATCH case value:

- Put a DISPLAY FORM or USE FORM instruction in all case values except the ANYMATCH

- Put instructions in ANYMATCH that do not reference form fields
- Do not put a DISPLAY FORM or USE FORM instruction in the ANYMATCH case value

If you are not conditionally referencing forms, put the DISPLAY FORM or USE FORM in the base request.

6.4.2.4 Case Values When You Use More Than One Control Value -- If you have more than one CONTROL FIELD IS instruction, you also have more than one series of case values. You can use the same case value twice if it is under a different control field each time you use it.

If you select case values that are meaningful strings, the request will be clearer to the programmer.

6.4.3 Match Instructions in a CONTROL FIELD IS Instruction

Match instructions are request instructions to be executed when a case value matches the control value. You can associate any number of match instructions with a particular case value.

Each match instruction is a request instruction in itself and must therefore be followed by a semicolon (;).

When you create a conditional instruction, you can specify any request instruction following a case value (except the FORM IS and RECORD IS instructions), including instructions to:

- Display forms
- Input, output, and return data
- Change video attributes
- Use program request keys (discussed in Chapter 11, Program Request Keys)

Mapping Between Form Arrays and Record Arrays 7

This chapter describes how to map between form arrays (including scrolled form arrays) and record arrays.

The ability to define mappings between form and record arrays is a particularly powerful feature of the request. Once you define an array mapping, TDMS handles the display and collection of data within the scrolled or indexed form array. The application program, therefore, as in all other form I/O operations, need contain no code to deal with data in a scrolled form region.

The examples in this chapter illustrate the correct syntax for referencing form arrays and record arrays when you create mappings. They also show the kinds of array mappings you may wish to use in a TDMS application.

7.1 What Is an Array?

An **array** (either a record or a form array) is a table of items in which all the items are referred to by the same name and have the same characteristics.

For example, there are four elements in the record or form array LAST_NAME in Figure 7-1. You refer to an element by using the name of the array and a number (**subscript**) that indicates the position of the element in the array. The array LAST_NAME begins at 1 and has the subscripts 1, 2, 3, and 4. To refer to the third element of the array LAST_NAME, you use the array name and the number 3.

A form array always begins at 1. A record array defined using CDDL can begin at any value. If a record array begins with a subscript other than 1, you count from the lowest subscript. Suppose a record array begins at 0, and you want to refer to the third element of that array. Because the array begins at 0, the subscripts are 0, 1, 2, and 3, so you use the array name and the number 2.

See the section entitled Explicit Syntax for Mapping Array Elements for more information about array subscripts.

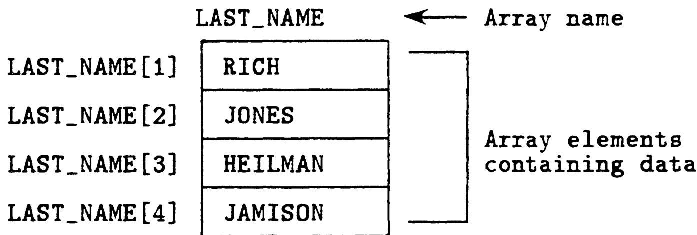


Figure 7-1: Definition of an Array

Note that each element of an array can contain a unique piece of data that has the same characteristics as all the other elements of the array. Figure 7-1 shows the array LAST_NAME, where all the elements of the array contain different last names. Each last name consists of alphabetic data with a maximum size of 7 characters.

The array in Figure 7-1 is a one-dimensional array. That is, it consists of a single row of data, and each element in that array is referred to by a single subscript value. Each element in that array has the same data type and the same maximum field length. Chapter 8, Advanced Mapping Between Arrays, discusses the use of two-dimensional arrays in TDMS requests.

7.1.1 Types of Form Arrays You Can Map to and from

You can create mappings to indexed, scrolled, and horizontally-indexed scrolled form arrays in TDMS. This chapter discusses indexed and scrolled arrays. Chapter 8, Advanced Mapping Between Arrays, discusses mapping horizontally-indexed scrolled arrays.

Both indexed and scrolled form arrays are one-dimensional and you refer to elements in them with a single subscript. The only difference between the two form array types is that with indexed form arrays you can map only a fixed number of elements, while with scrolled form arrays you can map an almost unlimited number of elements.

- In an **indexed form array**, you designate how many elements the array contains. An indexed array can be horizontally or vertically indexed.

  For example, the form in Figure 7-2 contains a single, vertically-indexed array, LAST_NAME. The array contains four elements, each of which has the same name, length, and data type. As the example shows, each element can contain a separate piece of data.

Form Definition
Indexed Form Array

```
LAST_NAME:
        AAAAAAA
        AAAAAAA
        AAAAAAA
        AAAAAAA
```

Run-Time Form
Indexed Form Array

```
LAST_NAME:
        RICH
        JONES
        HEILMAN
        JAMISON
```

Figure 7-2: Indexed Array

- In a scrolled form array or region, the total number of elements is not defined when the form is created. Instead, a **window** into the scrolled array is defined.

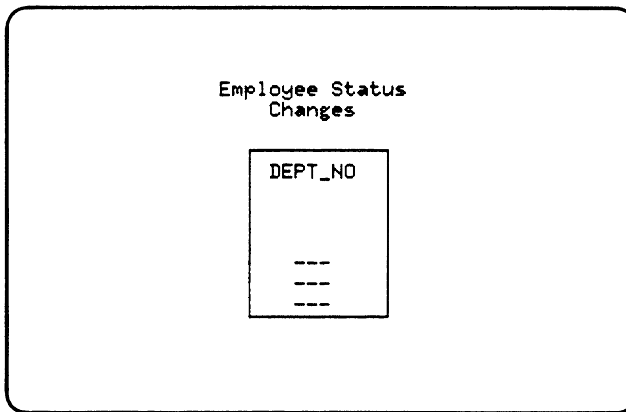
  A window is one or more lines defined as a scrolled region on a form. At run time, the operator can scroll data up and down within the window to see or enter much more than one line of data.

For example, in Figure 7-3, DEPT_NO is a scrolled array with a window of three lines. You can map the seven elements in the record array WK_DEPT to the scrolled form array DEPT_NO. At run time, if the operator moves through the scrolled window, TDMS displays all seven elements, three at a time, in the three-line window.

See Chapter 10, How to Display and Input Data in a Scrolled Region, for information on how data is presented in a scrolled array.

When you create a form with one or more scrolled form arrays, the entire width of the form that contains the form array is called a scrolled region. You define mappings to several scrolled arrays in exactly the same way as you do for a single scrolled array.

Scrolled Form



← Only three lines are visible at any one time in the form window

Record Array Within Group Field

```
DEFINE RECORD PERSONNEL_RECORD.
 PERSONNEL STRUCTURE.
   WK_DEPT OCCURS 7 TIMES DATATYPE TEXT 3.
 END PERSONNEL STRUCTURE.
END PERSONNEL_RECORD.
```

← Seven elements are mapped to the scrolled form array

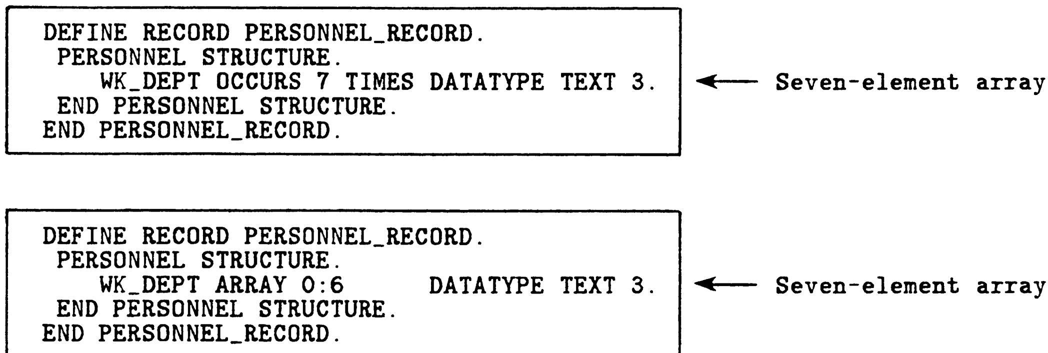
Figure 7-3: Scrolled Array

7.1.2 Types of Record Arrays You Can Map to and from

TDMS lets you map form arrays to and from one-dimensional and two-dimensional record arrays. This chapter illustrates one-dimensional record arrays.

A one-dimensional record array, in TDMS, is any array whose elements can be identified by a single subscript value. Figures 7-4 and 7-5 contain examples of two types of one-dimensional record arrays, simple arrays and group arrays.

In a **simple array**, each element has the same name, data type, length, and other field characteristics. A simple array has no subfields. For example, in Figure 7-4, WK_DEPT is a simple array that contains 7 elements. You refer to a simple array using the array name and a subscript value.

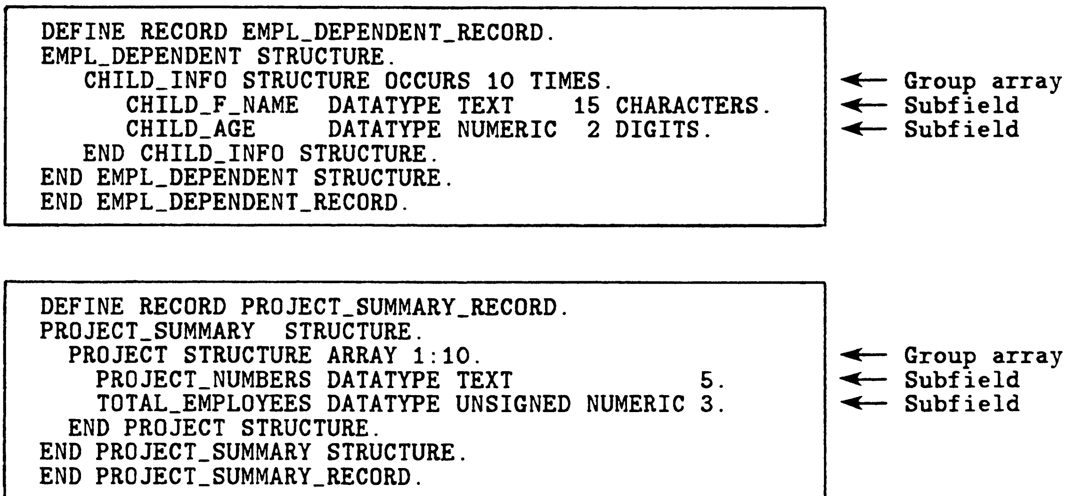


```
DEFINE RECORD PERSONNEL_RECORD.
 PERSONNEL STRUCTURE.
    WK_DEPT OCCURS 7 TIMES DATATYPE TEXT 3.     <--- Seven-element array
 END PERSONNEL STRUCTURE.
END PERSONNEL_RECORD.
```

```
DEFINE RECORD PERSONNEL_RECORD.
 PERSONNEL STRUCTURE.
    WK_DEPT ARRAY 0:6      DATATYPE TEXT 3.     <--- Seven-element array
 END PERSONNEL STRUCTURE.
END PERSONNEL_RECORD.
```

Figure 7-4: Simple One-Dimensional Record Arrays

In a **group array**, each element of the array contains other fields, called subfields. For instance, in Figure 7-5, the group array CHILD_INFO repeats 10 times. Each occurrence of CHILD_INFO contains the subfields CHILD_F_NAME and CHILD_AGE. You refer to a particular subfield entry in the array by the subfield name and a single subscript.



```
DEFINE RECORD EMPL_DEPENDENT_RECORD.
EMPL_DEPENDENT STRUCTURE.
   CHILD_INFO STRUCTURE OCCURS 10 TIMES.                 <--- Group array
      CHILD_F_NAME  DATATYPE TEXT    15 CHARACTERS.      <--- Subfield
      CHILD_AGE     DATATYPE NUMERIC  2 DIGITS.          <--- Subfield
   END CHILD_INFO STRUCTURE.
END EMPL_DEPENDENT STRUCTURE.
END EMPL_DEPENDENT_RECORD.
```

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY  STRUCTURE.
  PROJECT STRUCTURE ARRAY 1:10.                          <--- Group array
    PROJECT_NUMBERS DATATYPE TEXT                5.      <--- Subfield
    TOTAL_EMPLOYEES DATATYPE UNSIGNED NUMERIC 3.         <--- Subfield
  END PROJECT STRUCTURE.
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_SUMMARY_RECORD.
```

Figure 7-5: One-Dimensional Group Arrays

Both the group array and its subfields are referred to as arrays. For instance, we say the CHILD_AGE array has 10 elements and that the group array CHILD_INFO has 10 elements.

In the preceding examples, the record definitions contain the two keywords CDDL uses to define arrays, OCCURS and ARRAY:

- The keyword OCCURS and the number following it identify the field as an array and indicate the number of times the elements in the array repeat.
- The keyword ARRAY and the numbers following it identify the field as an array and indicate the upper and lower limits of the array subscript.

The syntax you use to map arrays is the same whether your request uses a record containing the OCCURS syntax or a record containing the ARRAY syntax. See the *VAX Common Data Dictionary Data Definition Language Reference Manual* for more information on record definition syntax. See Chapter 16, Using Record Definitions, for more information on how programming languages convert these arrays for use in an application program.

7.2 Syntax for Mapping Between Form and Record Arrays

You can map arrays using either explicit syntax (with subscript values) or the %ALL mapping syntax. Figure 7-6 shows array mappings with both types of syntax.

Request

```
FORM IS          EMPLOYEE_STATUS_CHANGES;
RECORDS ARE      EMPL_DEPENDENT_RECORD,
                 PERSONNEL_RECORD;

DISPLAY FORM     EMPLOYEE_STATUS_CHANGES;

  OUTPUT WK_DEPT[3] TO DEPT_NO[1 TO 5];          <--- Maps element 3 of a simple
                                                      record array to elements
                                                      1 to 5 of the form array

  OUTPUT CHILD_AGE[1 TO 10] TO                   <--- Maps elements 1 to 10 of
              CHILD_AGE[1 TO 10];                     the subfield CHILD_AGE to
                                                      the corresponding elements
                                                      of the scrolled form array

  INPUT %ALL;                                    <--- Inputs all the elements
                                                      of form arrays CHILD_AGE
END DEFINITION;                                       and DEPT_NO
```

Figure 7-6: Example of Mapping Arrays

7.2.1 Explicit Syntax for Mapping Array Elements

If you explicitly map array elements, the subscript values identify the elements that you wish to map.

You can explicitly map:

- A single element in a form or record array:

  ```
  CHILD_AGE[3]
  ```

- All the elements in a form or record array:

  ```
  CHILD_AGE[1 TO 10]
  ```

- A subset of elements in a form or record array:

  ```
  CHILD_AGE[2 TO 6]
  ```

The subscript always follows the array name and is enclosed in brackets. When you refer to subfields in group arrays, the subscript value always follows the final subfield in the reference, rather than the group array name. For example, the following subfield name and subscript value refers to the third element of the subfield CHILD_AGE:

```
CHILD_INFO.CHILD_AGE[3]
```

The subscript must be a positive integer constant. If you refer to the first element of an array, the subscript you use must be a 1; it cannot be a 0. The first element in a form array always has a subscript of 1. However, when a record is defined using CDDL or VAX DATATRIEVE, the first subscript can have any value.

If a request uses a CDD record array that has a first subscript value other than 1, RDU redefines the array to a one-based array. For example, if you create a request that refers to a zero-based record array, RDU displays the following message indicating it is adjusting the bounds of the array:

```
          WK_DEPT   ARRAY 0:6
0008
.............................1
%RDU-W-CHNGBND, changing the bound of the array from 0:6 to 1:7
```

RDU redefines the elements of the array WK_DEPT:

```
WK_DEPT[0] -->  WK_DEPT[1]
WK_DEPT[1] -->  WK_DEPT[2]
       .
       .
       .
WK_DEPT[6] -->  WK_DEPT[7]
```

When you reference the first element in your request, you use the subscript [1]. If you do not, RDU tells you that your reference is outside the bounds of the array.

This conversion does not change the array itself or the number of elements in the array. WK_DEPT still has seven elements. Although you access them by using the subscripts [1] to [7], the application program references them according to the normal procedure of the program language. For instance, BASIC would reference WK_DEPT[0 TO 6].

7.2.2 %ALL Syntax for Mapping Array Elements

If you use the %ALL syntax to refer to all the elements of a form and a record array, you do not explicitly specify subscripts (or array names).

Note that when you use %ALL to map:

- A *scrolled* form array, RDU maps the number of elements in the record array (since there is no defined size for a scrolled form array).
- An *indexed* form array, RDU attempts to map the smaller number of elements, whether they are in the record or the form array.

This is true whether the mappings are INPUT TO, OUTPUT TO, or RETURN TO instructions.

As you would expect, when RDU creates mappings for an entire form or record array, it begins by creating a mapping between the first element of the sending and receiving arrays. It maps that first element of the sending form or record array to the first element of the receiving array. It then continues to map each succeeding element to the corresponding element in the receiving array. For example, the INPUT %ALL instruction in Figure 7-6 results in the following run-time mappings:

```
Inputs CHILD_AGE[1] TO CHILD_AGE[1]
       CHILD_AGE[2] TO CHILD_AGE[2]
                   .
                   .
                   .
       CHILD_AGE[10] TO CHILD_AGE[10]
```

The first element mapped in both form and record arrays is always assigned the subscript value of [1]. As with explicit mappings, if the first element in a CDD record array has a subscript value other than [1], RDU redefines this array to be a one-based array.

7.3 Rules for Mapping Arrays

The following rules apply to all mappings you create between form and record arrays (including two-dimensional arrays):

- You can map a single element of an array to a single element of any other array (or to a simple field):

  ```
  OUTPUT SIMPLE_RECORD_FIELD TO SCROLLED_FORM_ARRAY[3];
  INPUT SCROLLED_FORM_ARRAY[3] TO TWO_DIM_ARRAY[3,4];
  ```

- You can map a single element of a sending array to a range of elements in a receiving array:

  ```
  OUTPUT RECORD_ARRAY[1] TO INDEXED_FORM_ARRAY[3 TO 5];
  INPUT SCROLLED_FORM_ARRAY[3] TO TWO_DIM_ARRAY[3, 4 TO 5];
  ```

- You can map a range of elements to a range of elements:

  ```
  OUTPUT RECORD_ARRAY[3 TO 5] TO SCROLLED_FORM_ARRAY[3 TO 5];
  INPUT SCROLLED_FORM_ARRAY[2 TO 7] TO RECORD_ARRAY[1 TO 6];
  ```

The maximum number of dimensions for a record field is 16.

If you *explicitly* map more than a single element from a sending array, these additional rules apply:

- The receiving array and sending array must have the same number of dimensions. For example, since scrolled and indexed form arrays have only one dimension, you must map them to record arrays of only one dimension.
- The mapping instruction must specify the same number of elements in each corresponding range in the sending and receiving arrays. You cannot explicitly map a smaller (or larger) range of elements to a larger (or smaller) range of elements.

  If RECORD_ARRAY contains only four elements and FORM_ARRAY contains ten elements, the following mapping instruction is *incorrect* because the range in the sending array is smaller than the corresponding range in the receiving array:

  ```
  OUTPUT RECORD_ARRAY[1 TO 4] TO FORM_ARRAY[1 TO 10];
  ```

  However, you can use a subset of the larger FORM_ARRAY and create a correct mapping:

  ```
  OUTPUT RECORD_ARRAY[1 TO 4] TO FORM_ARRAY[1 TO 4];
  INPUT FORM_ARRAY[3 TO 6] TO RECORD_ARRAY[1 TO 4];
  ```

- The elements you specify in a form and record array reference must be within the bounds (in each dimension) of the form and record array definitions.

7.3.1 Explicit Mappings and Errors

In explicit mappings, RDU displays error level messages and does not create a request if you do not follow the array mapping rules specified in the previous section.

7.3.2 %ALL Mappings and Errors

In a %ALL mapping, you are implicitly mapping an entire array. The number of elements in the receiving and sending arrays do not have to match. In the following portion of a request, RDU maps all those elements for which it finds matching receiving elements. Note that the record and form arrays have different defined sizes.

```
                        .
                        .
                        .
INPUT %ALL;
OUTPUT %ALL;
DESCRIPTION /* The record array contains 4 elements.
     The scrolled form array contains 10 elements.

     INPUT %ALL
            inputs form array ARRAY[1 TO 4] to
            record array ARRAY[1 TO 4]

     OUTPUT %ALL
            outputs record array ARRAY[1 TO 4] to
            form array ARRAY[1 TO 4] */;
                        .
                        .
                        .
```

When you use the %ALL syntax to map between a larger array and a smaller array, RDU always maps the smaller number of elements. RDU generates an information level message and indicates the elements it was able to map (if you specify the /LOG qualifier with the CREATE, MODIFY, REPLACE, or VALIDATE commands). If you specify the /LOG qualifier with the BUILD LIBRARY command, RDU indicates the elements it was able to map if:

- The request was created with the /NOSTORE qualifier
- The BUILD LIBRARY is validating each request in the library

As in requests with nonarray mappings, if RDU is able to create any of the mappings implied by %ALL, it creates the %ALL mapping successfully.

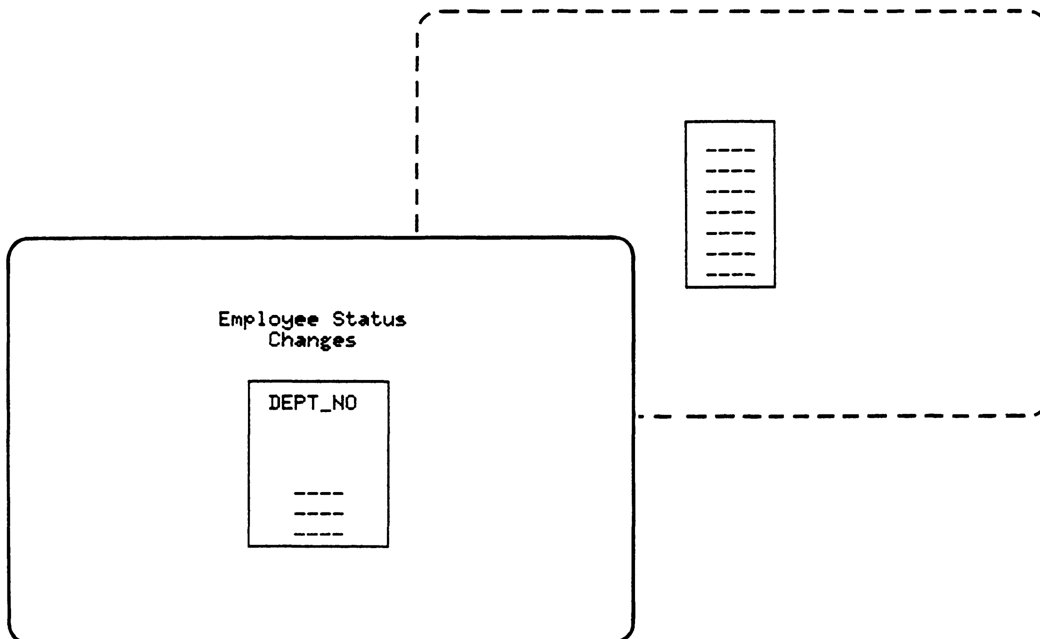
7.4 Examples of Mapping Indexed and Scrolled Arrays

The remaining sections of this chapter contain examples of mappings between indexed and scrolled form arrays and one-dimensional record arrays. Note that you use the same syntax to explicitly map indexed and scrolled form arrays. The only difference is:

- When you map an indexed array, you can map only the number of elements defined in the indexed form array definition
- When you map a scrolled array, you can map as many elements as are contained in the matching record array

When you map a record array to or from a scrolled form array, TDMS sets up an underlying form array structure to contain all the elements of the array you map. The operator does not see this array. It is the underlying structure that contains all the elements you map to or from the scrolled array.

For instance, in Figure 7-7, the underlying form array contains seven elements when you map the seven element record array DEPT_NO to the form array DEPT_NO.



(continued on next page)

Figure 7-7: The Underlying Form Array

One-Dimensional Record

```
DEFINE RECORD EMPLOYEE_HIST_RECORD.
EMPLOYEE_HIST STRUCTURE.
   DEPT_CHANGES STRUCTURE.
             OCCURS 7 TIMES.
      DEPT_NO  DATATYPE TEXT 4.
      DEPT_LOC DATATYPE TEXT 3.
   END DEPT_CHANGES STRUCTURE.
END EMPLOYEE_HIST STRUCTURE.
END EMPLOYEE_HIST_RECORD.
```

Figure 7-7: The Underlying Form Array (Cont.)

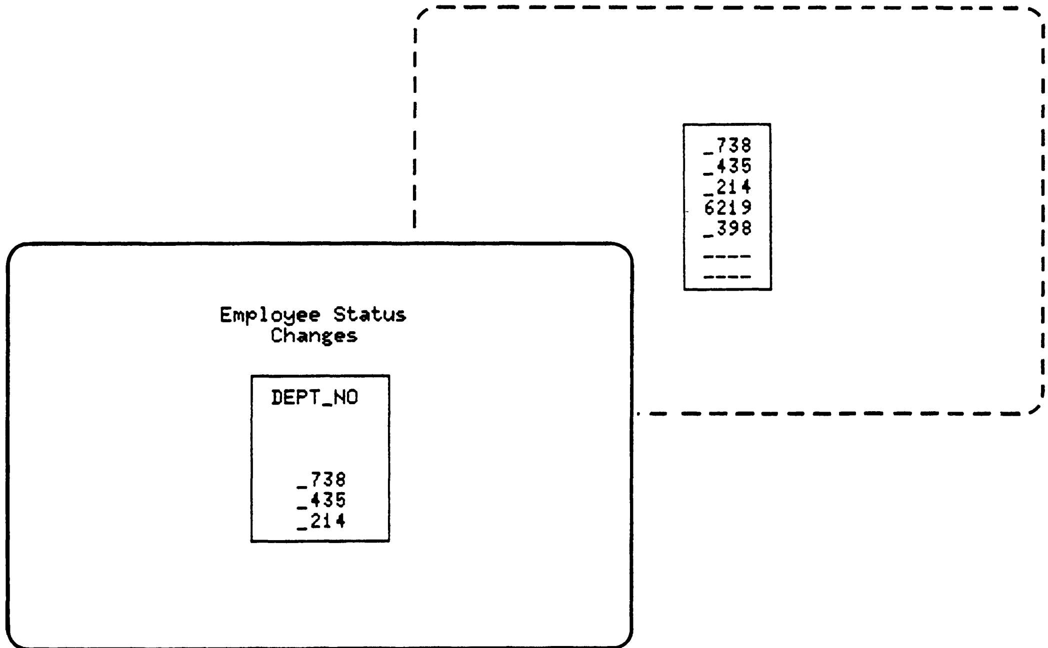
TDMS determines which elements of the underlying form array are displayed in the scrolled form array. See Chapter 10, How to Display and Input Data in a Scrolled Region, for a discussion of screen presentation of data in a scrolled region.

7.4.1 Explicit Mapping of Scrolled or Indexed Arrays

The request in Figure 7-8 shows how you can map an entire form array, DEPT_NO, to and from a group record array, DEPT_CHANGES, subfield DEPT_NO.

Note that, because a scrolled form array has no predefined size, the number of elements you specify in a mapping determines the number of elements in the form array. (The number of elements you specify cannot exceed the number of elements in the record array that you are mapping.) In the following example, for instance:

- The output mapping in the request establishes a size of seven elements for the scrolled form array
- The input mapping in the request establishes a size of five elements for the scrolled form array



Record

```
 DEFINE RECORD EMPLOYEE_HIST_RECORD.
 EMPLOYEE_HIST STRUCTURE.
     DEPT_CHANGES STRUCTURE
            OCCURS 7 TIMES.
         DEPT_NO  DATATYPE TEXT 4.
         DEPT_LOC DATATYPE TEXT 3.
      END DEPT_CHANGES STRUCTURE.
END EMPLOYEE_HIST STRUCTURE.
END EMPLOYEE_HIST_RECORD.
```

Request

```
FORM IS      EMPLOYEE_STATUS_CHANGES;
RECORD IS    EMPLOYEE_HIST_RECORD;

DISPLAY FORM EMPLOYEE_STATUS_CHANGES;

    OUTPUT DEPT_NO[1 TO 7] TO DEPT_NO[1 TO 7];    <--- Seven elements
                                                       mapped for output

    INPUT DEPT_NO[1 TO 5] TO DEPT_NO[1 TO 5];     <--- Five elements
                                                       mapped for input
END DEFINITION;
```

Figure 7-8: Explicitly Mapping an Entire Scrolled Form Array

7.4.2 %ALL Mappings

You can also map all the elements to and from a scrolled or indexed form array using the %ALL syntax. In this section, you see how RDU creates %ALL mappings first for a scrolled form array and second for an indexed form array.

7.4.2.1 %ALL Mapping and a Scrolled Array -- Figure 7-9 shows a request that contains a %ALL mapping to and from a scrolled array. Note that because the scrolled array has no predefined size and because you do not specify subscripts with %ALL, RDU determines the number of elements to map.

Since the record array DEPT_NO contains seven elements, RDU creates input and output mappings for seven elements.

At run time, TDMS:

- Displays seven elements in the scrolled array DEPT_NO
- Collects seven elements from the operator in the scrolled array DEPT_NO

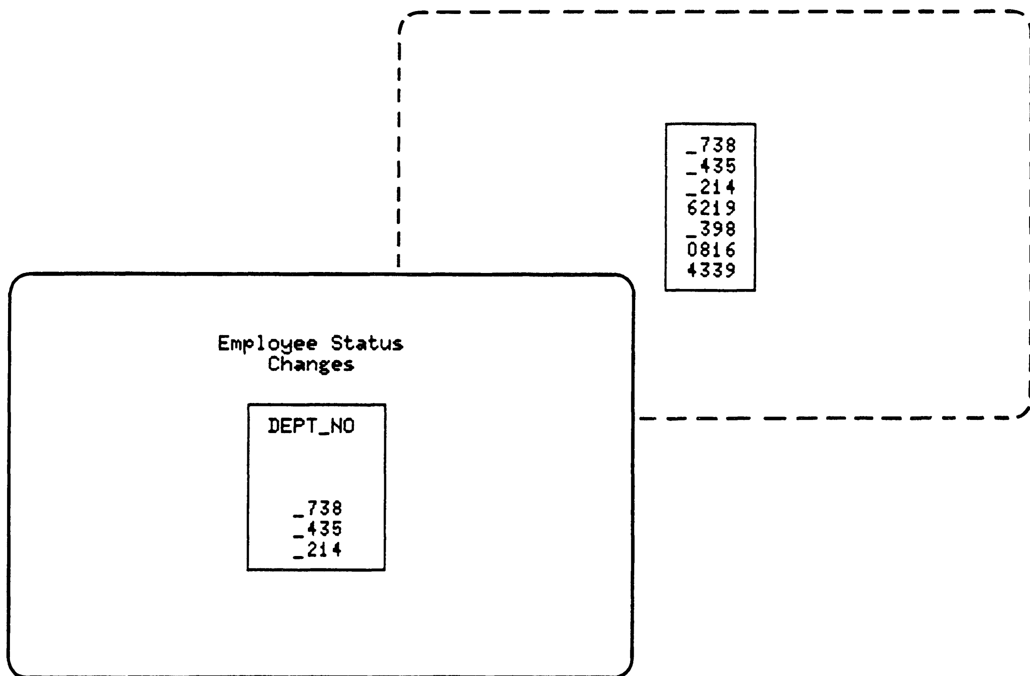


Figure 7-9: Using %ALL to Map an Entire Scrolled Array

One-Dimensional Record Array

```
 DEFINE RECORD EMPLOYEE_HIST_RECORD.
 EMPLOYEE_HIST STRUCTURE.
    DEPT_CHANGES STRUCTURE
            OCCURS 7 TIMES.
       DEPT_NO  DATATYPE TEXT 4.
       DEPT_LOC DATATYPE TEXT 3.
    END DEPT_CHANGES STRUCTURE.
END EMPLOYEE_HIST STRUCTURE.
END EMPLOYEE_HIST_RECORD.
```

Request

```
FORM IS      EMPLOYEE_STATUS_CHANGES;
RECORD IS    EMPLOYEE_HIST_RECORD;

DISPLAY FORM EMPLOYEE_STATUS_CHANGES;

  INPUT %ALL;
  OUTPUT %ALL;

END DEFINITION;
```

Figure 7-9: Using %ALL to Map an Entire Scrolled Array (Cont.)

7.4.2.2 Using %ALL Mappings and Indexed Arrays -- Figure 7-10 shows that you can use %ALL to map all the elements in an indexed array. Again, because you do not specify the number of elements in a %ALL mapping, RDU determines the number of elements to map when you create the request.

In the case of an indexed form array, RDU always attempts to map the smaller number of elements, whether in the indexed form array or the record array. This is true whether the mappings are INPUT TO, OUTPUT TO, or RETURN TO instructions.

The request in Figure 7-10 maps form arrays that each contain one less element than the record array.

At run time, TDMS:

- Inputs all four elements of the indexed form arrays PROJECT_NO, HOURS, and STATUS to corresponding elements in an identically named record array
- Outputs four out of the five elements in the record arrays PROJECT_NO, HOURS, and STATUS to the corresponding elements in the identically named form arrays

Form

```
Employee Project Form

EMPLOYEE_NAME: __________

PROJECT_NO      HOURS    STATUS

----------       ---       --        <--- Four
----------       ---       --             indexed
----------       ---       --             elements
----------       ---       --
```

Group Record Array

```
DEFINE RECORD PROJECT_RECORD.
PROJECT STRUCTURE.
  EMPLOYEE_NAME DATATYPE TEXT 1.
  PRO_INFO STRUCTURE OCCURS              <--- Five elements
               5 TIMES.
    PROJECT_NO DATATYPE TEXT    10
    HOURS      DATATYPE NUMERIC 3.
    STATUS     DATATYPE TEXT    2.
  END PRO_INFO STRUCTURE.
END PROJECT STRUCTURE.
END PROJECT_RECORD.
```

Request

```
FORM IS       PROJECT_FORM;
RECORD IS     PROJECT_RECORD;

DISPLAY FORM PROJECT_FORM;

  INPUT %ALL;
  OUTPUT %ALL;

END DEFINITION;
```

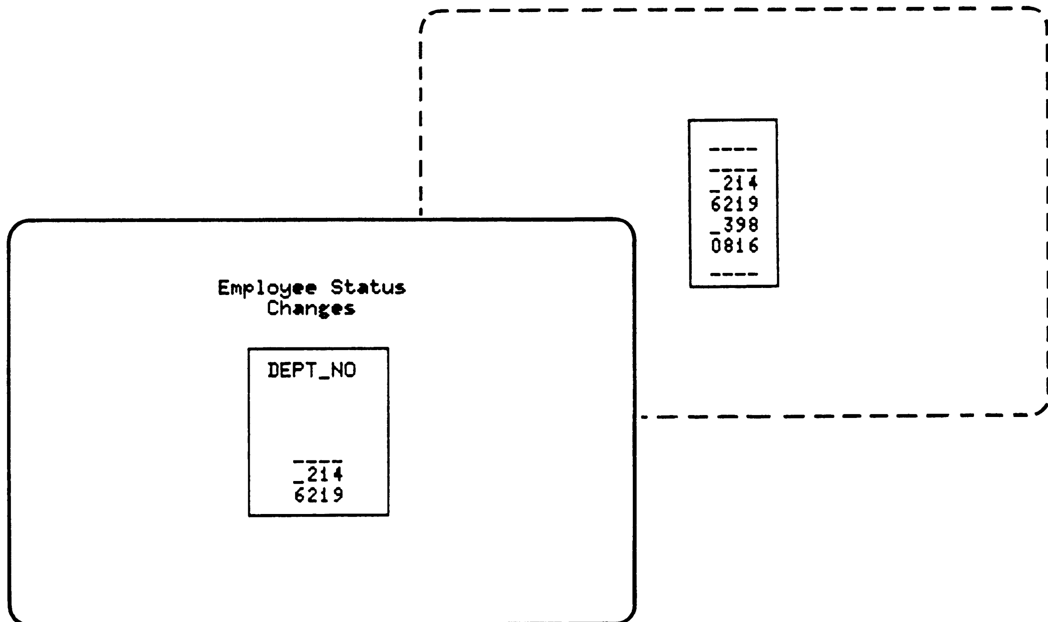
Figure 7-10: Using %ALL to Map Entire Indexed Arrays

Note that RDU always maps the smaller number of elements. RDU generates an information level message indicating which elements it was able to map if you use the /LOG qualifier.

7.4.3 Explicitly Mapping a Subset of a Scrolled or Indexed Array

You can map a subset of the entire array by using the range subscript. For both indexed form arrays and scrolled form arrays, the rules for mapping an entire array apply.

For example, in Figure 7-11, the request maps elements 3 through 7 of the group record array DEPT_NO, to elements 2 through 6 of the form array DEPT_NO.



One-Dimensional Record Array

```
DEFINE RECORD EMPLOYEE_HIST_RECORD.
EMPLOYEE_HIST STRUCTURE.
   DEPT_CHANGES STRUCTURE
            OCCURS 7 TIMES.
      DEPT_NO  DATATYPE TEXT 4.
      DEPT_LOC DATATYPE TEXT 3.
   END DEPT_CHANGES STRUCTURE.
END EMPLOYEE_HIST STRUCTURE.
END EMPLOYEE_HIST_RECORD.
```

Request

```
FORM IS      EMPLOYEE_STATUS_CHANGES;
RECORD IS    EMPLOYEE_HIST_RECORD;

DISPLAY FORM EMPLOYEE_STATUS_CHANGES;
```

(continued on next page)

Figure 7-11: Mapping a Subset of an Indexed or Scrolled Array

```
   OUTPUT DEPT_NO[3 TO 7] TO DEPT_NO[2 TO 6];
   INPUT  DEPT_NO[2 TO 6] TO DEPT_NO[3 TO 7];

END DEFINITION;
```

Figure 7-11: Mapping a Subset of an Indexed or Scrolled Array (Cont.)

7.4.4 Mapping Scrolled Arrays to Several Record Arrays

You can map to and from several scrolled (or indexed) arrays just as you do from a single scrolled array.

You can map these arrays to one or more record definitions. This section illustrates mapping several form arrays to two records: a group array and a simple array.

7.4.4.1 Explicitly Mapping Several Scrolled Arrays -- Figure 7-12 maps all elements of the CHILD_NAME array to two scrolled form arrays, LAST and FIRST. It also maps the elements in the CHILD_AGE simple array to the scrolled form array CHILD_AGE.

Note that the CHILD_NAME record array is a group record array. Because it contains two subfields, LAST and FIRST, you can map two scrolled or indexed form arrays to and from this record array.

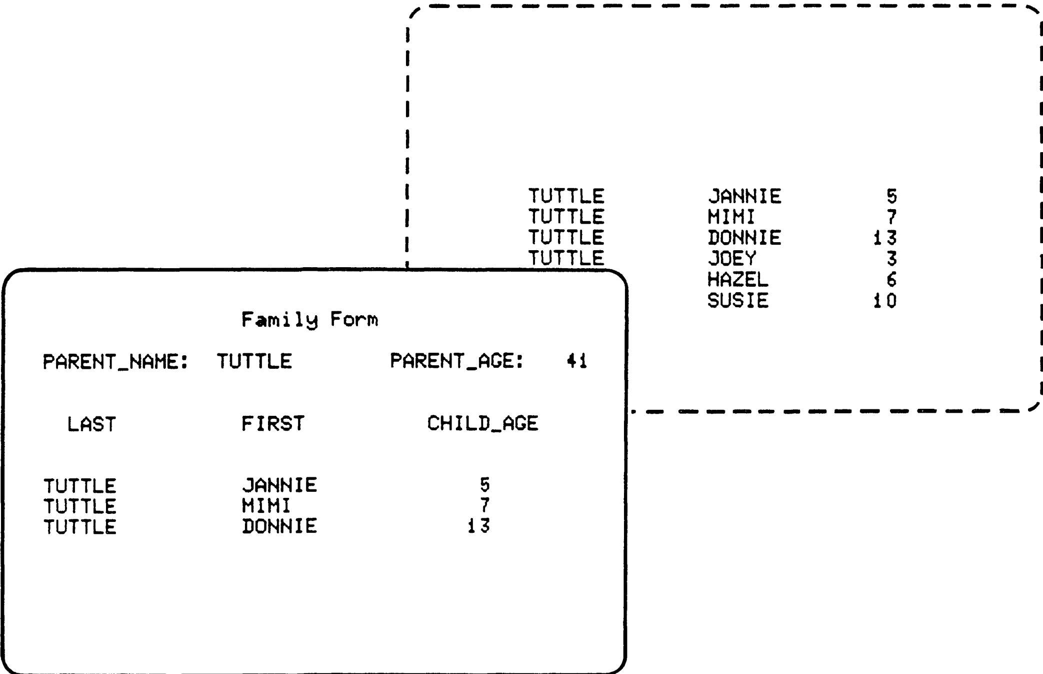


Figure 7-12: Explicit Mapping of Several Scrolled Arrays

One-Dimensional Group Record Array

```
DEFINE RECORD FAML_NAME_RECORD.
FAML_NAME STRUCTURE.
  PARENT_NAME DATATYPE    TEXT    10.
  CHILD_NAME STRUCTURE                    <--- Group array
        OCCURS 6 TIMES.                        CHILD_NAME
     LAST     DATATYPE    TEXT    10.
     FIRST    DATATYPE    TEXT    10.
  END CHILD_NAME STRUCTURE.
END FAML_NAME  STRUCTURE.
END FAML_NAME_RECORD.
```

One-Dimensional Simple Array

```
DEFINE RECORD FAML_AGE_RECORD.
FAML_AGE STRUCTURE.
 PARENT_AGE   DATATYPE    NUMERIC 2.
 CHILD_AGE OCCURS 6 TIMES                 <--- Simple array
             DATATYPE    NUMERIC 2.            CHILD_AGE
END FAML_AGE STRUCTURE.
END FAML_AGE_RECORD.
```

Request

```
FORM   IS FAMILY_FORM;
RECORD IS FAML_NAME_RECORD;
RECORD IS FAML_AGE_RECORD;

DISPLAY FORM FAMILY_FORM;

 OUTPUT
   PARENT_NAME TO PARENT_NAME,
   PARENT_AGE  TO PARENT_AGE;
 OUTPUT
    LAST[1 TO 6]  TO LAST[1 TO 6],        <--- Maps six
    FIRST[1 TO 6] TO FIRST[1 TO 6],            elements
    CHILD_AGE[1 TO 6] TO CHILD_AGE[1 TO 6];
 WAIT;
END DEFINITION;
```

Figure 7-12: Explicit Mapping of Several Scrolled Arrays (Cont.)

You define the size of the scrolled form arrays when you explicitly state the subscript ranges. In addition, although you are referring to record fields in two separate record definitions, the record names are not needed. All the field names, FIRST, LAST, CHILD_AGE, PARENT_NAME, and PARENT_AGE, are unique within the record definitions (FAML_NAME_RECORD and FAML_AGE_RECORD) used by the request.

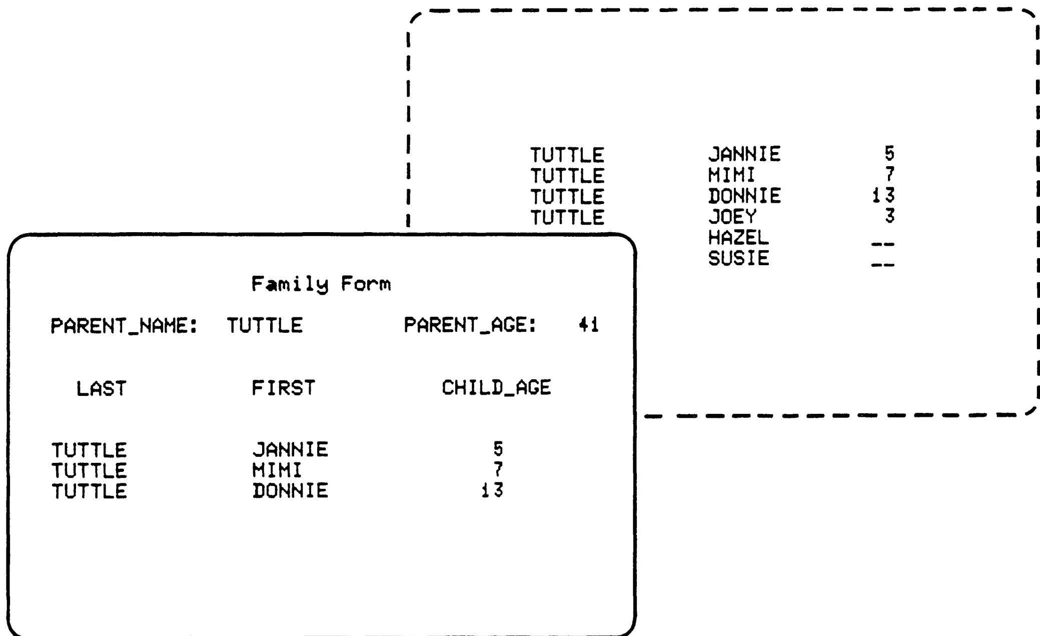
7.4.4.2 %ALL Mapping of Several Scrolled Arrays

7.4.4.2 %ALL Mapping of Several Scrolled Arrays -- Figure 7-13 maps three scrolled arrays to one-dimensional simple and group record arrays using the %ALL syntax. As with any %ALL array mappings, you do not specify the elements to be mapped. When you create the request, RDU determines the number of elements to map depending on the size of the record arrays.

In this example, the record array CHILD_AGE has four elements, while the record array CHILD_NAME has six elements. RDU, therefore, creates:

- Six input and output mappings for the scrolled arrays LAST and FIRST
- Four input and output mappings for the scrolled array CHILD_AGE

Although the record elements are mapped to three scrolled arrays that have the same window, you do not have to map the same number of elements to each scrolled array.



One-Dimensional Group Record Array

```
DEFINE RECORD FAML_NAME_RECORD.
FAML_NAME STRUCTURE.
   PARENT_NAME  DATATYPE TEXT 10.
   CHILD_NAME STRUCTURE OCCURS 6 TIMES.
       LAST     DATATYPE TEXT 10.
       FIRST    DATATYPE TEXT 10.
   END CHILD_NAME STRUCTURE.
END FAML_NAME  STRUCTURE.
END FAML_NAME_RECORD.
```

← Six-element group array CHILD_NAME

Figure 7-13: %ALL Mapping of Several Scrolled Arrays

One-Dimensional Simple Array

```
DEFINE RECORD FAML_AGE_RECORD.
FAML_AGE STRUCTURE.
   PARENT_AGE   DATATYPE NUMERIC  2.
   CHILD_AGE OCCURS 4 TIMES
                DATATYPE NUMERIC  2.
END FAML_AGE STRUCTURE.
END FAML_AGE RECORD.
```

← Four-element simple array CHILD_AGE

Request

```
FORM IS      FAMILY_FORM;
RECORD IS    FAML_NAME_RECORD;
RECORD IS    FAML_AGE_RECORD;

DISPLAY FORM FAMILY_FORM;

   INPUT %ALL;
   OUTPUT %ALL;

END DEFINITION;
```

Figure 7-13: %ALL Mapping of Several Scrolled Arrays (Cont.)

Advanced Mapping Between Arrays 8

The preceding chapter discussed mapping between indexed and scrolled form arrays and one-dimensional record arrays. This chapter explains how to map between a horizontally-indexed scrolled form array and two-dimensional record arrays.

8.1 Horizontally-Indexed Scrolled Form Arrays

A **horizontally-indexed form array** is an array with several elements that repeat across a form. For instance, the form array CHILD_NAME in Figure 8-1 repeats three times across the line on a form.

A horizontally-indexed array can also be a scrolled array. In Figure 8-1, for example, the array CHILD_NAME consists of an undefined number of scrolled rows as well as the three indexed columns. (Note that PARENT_LAST_NAME is a separate one-dimensional scrolled array on the form.)

```
        Employee Dependent Form

PARENT_LAST_NAME      CHILD_NAME

---------------       -----  -----  -----
---------------       -----  -----  -----
       .                .
       .                .
       .                .
---------------       -----  -----  -----
```

Figure 8-1: Horizontally-Indexed Scrolled Form Array

A horizontally-indexed scrolled array always has two dimensions:

- The first dimension, the scrolled dimension, has no fixed size when the form is created.
- The second dimension, the horizontally-indexed portion of the array, has a fixed number of elements when the form is created.

8.2 Two-Dimensional Record Arrays

A **two-dimensional record array** is any array that contains elements that are specified using two subscript values.

You can map a one-dimensional form array to:

- A one-dimensional record array
- A two-dimensional record array

Note

> In TDMS, arrays are categorized as one-dimensional or two-dimensional according to whether you refer to the elements in those arrays by one or two subscript values, respectively. How a particular programming language refers to these same fields does not affect how you refer to them in a request. See Chapter 16, Using Record Definitions, for information on how BASIC, FORTRAN, and COBOL convert these arrays to record definitions the programs can use.

The following examples show two-dimensional arrays.

These record definitions were created using CDDL. As in the preceding chapter, both the ARRAY and OCCURS syntax are shown:

- Where the keywords OCCURS or ARRAY are nested, the inner, or nested OCCURS or ARRAY indicates the second dimension of the array.
- Where the keyword ARRAY is not nested and two sets of subscripts are specified, the second set of subscript values identifies the second dimension of the array.

In FAMILY_RECORD, the array CHILD_NAME repeats three times within the group array FAMILY_NAME, which occurs five times.

FAMILY_RECORD

```
DEFINE RECORD FAMILY_RECORD.
FAMILY STRUCTURE.
   FAMILY_NUMBER
             DATATYPE NUMERIC  5 DIGITS.
   FAMILY_NAME STRUCTURE
       OCCURS 5 TIMES.
          PARENT_LAST_NAME
             DATATYPE TEXT    15.
          CHILD_NAME OCCURS 3 TIMES        <--- Nested OCCURS
             DATATYPE TEXT     5.               two-dimensional
   END FAMILY_NAME STRUCTURE.                   group array
END FAMILY STRUCTURE.
END FAMILY_RECORD.
```

In PROJECT_SUMMARY_RECORD, the array WAGE_CLASS repeats three times within the ten-element group array PROJECT.

PROJECT_SUMMARY_RECORD

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY  STRUCTURE.
   PROJECT STRUCTURE ARRAY 1:10.
      PROJECT_NUMBERS
            DATATYPE TEXT    5.
      TOTAL_EMPLOYEES
            DATATYPE NUMERIC 5.
      WAGE_CLASS ARRAY 1:3
            DATATYPE NUMERIC 5.              <--- Two-dimensional
   END PROJECT STRUCTURE.                         array
 END PROJECT_SUMMARY  STRUCTURE.
 END PROJECT_SUMMARY_RECORD.
```

In PROJECT_CONTROL_RECORD, the array CONTROL_PROJECT contains elements that repeat 30 times in a 10 by 3 matrix.

PROJECT_CONTROL_RECORD

```
DEFINE RECORD PROJECT_CONTROL_RECORD.
PROJECT_SUMMARY STRUCTURE.
  CONTROL_NO OCCURS 3 TIMES                  ---+
          DATATYPE NUMERIC 5.                   | One-dimensional
  CONTROL_COST ARRAY 1:5                        | arrays
          DATATYPE NUMERIC 5.                ---+
  CONTROL_PROJECT ARRAY 1:10,1:3             <--- Two-dimensional
          DATATYPE NUMERIC 5.                     array
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_CONTROL_RECORD.
```

8.3 Syntax for Mapping Two-Dimensional Arrays

With horizontally-indexed scrolled form arrays and two-dimensional record arrays, as with all arrays, you can refer to:

- A single element in the array
- A subset of all the elements using a range subscript
- The entire two-dimensional array using explicit subscript references
- All the elements of the array using the %ALL syntax

The subscript values you specify are positioned following the last record field or form field name you are mapping and are enclosed in brackets and separated by commas. For example:

```
FAMILY_NAME.CHILD_NAME[1 TO 5, 1 TO 3];
```

As in all references to record fields, you need specify only as many preceding group field names as necessary to make a record field reference unique. You can refer to CHILD_NAME in FAMILY_RECORD, for instance, as:

```
CHILD_NAME[1 TO 5, 1 TO 3];
```

8.4 General Rules for Two-Dimensional Arrays

With horizontally-indexed scrolled form arrays and two-dimensional record arrays:

- You can map a single element of an array to a single element of any other array (or to a simple field):

  ```
  OUTPUT TWO_DIM_ELEMENT[4,7] TO HORIZONTAL_ELEMENT[4,7];
  INPUT HORIZONTAL_ELEMENT[3,2] TO TWO_DIM_ELEMENT[3,2];
  ```

- You can map a single element of a sending array to a range of elements in a receiving array:

  ```
  OUTPUT TWO_DIM_ELEMENT[4,7] TO HORIZONTAL_RANGE[4 TO 7, 7 TO 10];
  INPUT HORIZONTAL_ELEMENT[3,2] TO TWO_DIM_RANGE[3, 2 TO 10];
  ```

- You can map a range of elements to a range of elements:

  ```
  OUTPUT TWO_DIM_RECORD_ARRAY[1 TO 8, 1 TO 4]
               TO HORIZONTAL_ARRAY[4 TO 11, 1 TO 4];
  INPUT  HORIZONTAL_ARRAY[3 TO 5, 2 TO 8]
               TO TWO_DIM_ARRAY[1 TO 3, 1 TO 7];
  ```

In all mappings, if you explicitly map more than a single element from a sending array:

- You must specify the same number of elements (in each dimension) in the receiving and sending arrays
- The sending and receiving arrays must have two dimensions
- The elements referenced in each dimension must be within the bounds (in each dimension) of the record and form array definitions

In a %ALL mapping of a two-dimensional array, you do not specify subscript values. In all cases where you are mapping two-dimensional arrays, both sending and receiving arrays must have two dimensions (unless both the arrays contain only a single element).

Because you do not specify the elements to map, when you create a request RDU determines the number of elements to map according to the following procedure:

- RDU maps to the horizontally-indexed area whichever is lower, either:
  - The number of elements in the indexed portion of the form array
  - The number of elements in the second dimension of the two-dimensional record array
- RDU maps to the scrolled area the number of elements in the first dimension of the record array

8.5 Examples of Mapping Two-Dimensional Arrays

The following sections show how you can map an entire horizontally-indexed scrolled array first using explicit syntax and then using %ALL. The final example shows how you can map a subset of a two-dimensional array.

8.5.1 Explicit Syntax to Map a Two-Dimensional Array

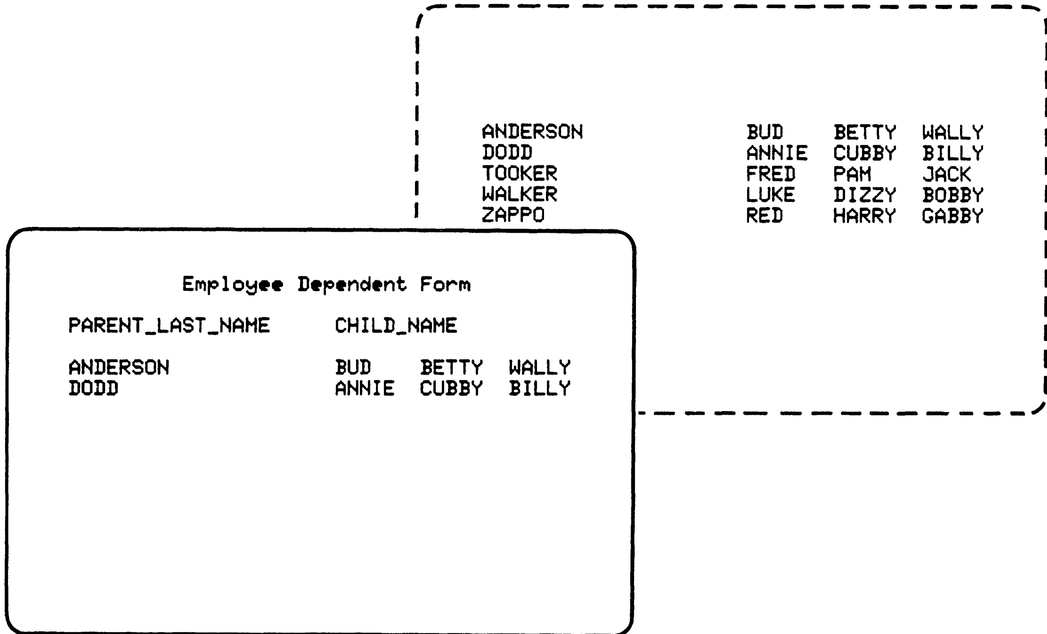
Figure 8-2 shows how you can explicitly specify a mapping to and from all the elements in a horizontally-indexed scrolled array.

The request maps for output:

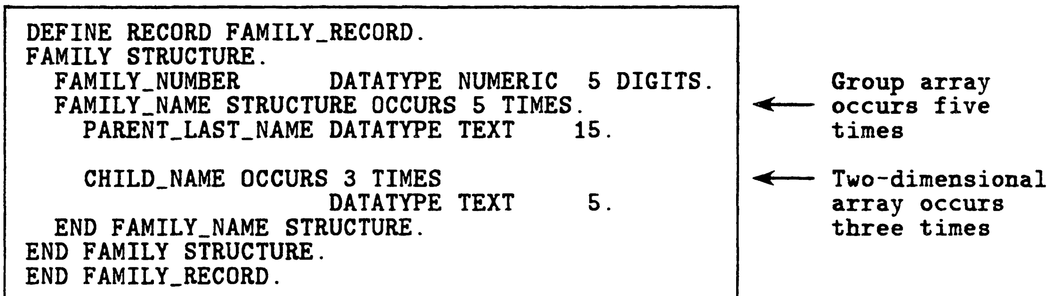
- All five elements in the one-dimensional array PARENT_LAST_NAME to the scrolled array PARENT_LAST_NAME
- All three elements in each occurrence of the two-dimensional record array CHILD_NAME to all three elements in each row of the horizontally-indexed scrolled form array CHILD_NAME

Notice in the mapping instructions that the scrolled or row portion subscript (1 to 5) of the CHILD_NAME array is expressed first, and the indexed or column subscript (1 to 3) is expressed last. The request maps the same number of elements (in each dimension) in the receiving and sending arrays.

EMPLOYEE_DEPENDENT_FORM



Two-Dimensional Group Record (FAMILY_RECORD)



```
DEFINE RECORD FAMILY_RECORD.
FAMILY STRUCTURE.
  FAMILY_NUMBER       DATATYPE NUMERIC  5 DIGITS.
  FAMILY_NAME STRUCTURE OCCURS 5 TIMES.
    PARENT_LAST_NAME DATATYPE TEXT    15.

    CHILD_NAME OCCURS 3 TIMES
                     DATATYPE TEXT     5.
  END FAMILY_NAME STRUCTURE.
END FAMILY STRUCTURE.
END FAMILY_RECORD.
```


Request

```
FORM IS   EMPLOYEE_DEPENDENT_FORM;
RECORD IS FAMILY_RECORD;

DISPLAY FORM EMPLOYEE_DEPENDENT_FORM;

   OUTPUT PARENT_LAST_NAME[1 TO 5]
            TO PARENT_LAST_NAME[1 TO 5],
```

Figure 8-2: Using Explicit Syntax to Map a Two-Dimensional Array

```
        CHILD_NAME[1 TO 5, 1 TO 3]
            TO CHILD_NAME[1 TO 5, 1 TO 3];
   WAIT;
END DEFINITION;
```

Figure 8-2: Using Explicit Syntax to Map a Two-Dimensional Array (Cont.)

8.5.2 Rules for %ALL Mapping of Two-Dimensional Arrays

In a %ALL mapping of a two-dimensional array, you do not specify subscript values. Because you do not explicitly specify the elements to map when you create the request, RDU determines the number of elements to map according to the following procedure:

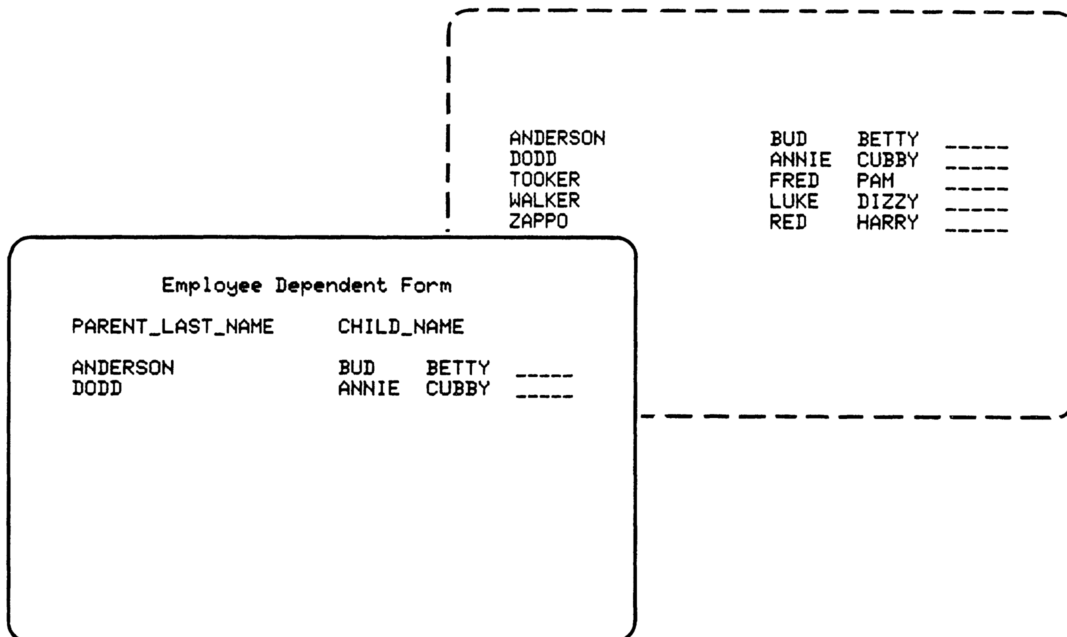
- RDU maps to the horizontally-indexed area whichever is lower, either:
    - The number of elements in the indexed portion of the form array
    - The number of elements in the second dimension of the two-dimensional record array
- RDU maps to the scrolled area the number of elements in the first dimension of the record array

8.5.3 %ALL to Map a Two-Dimensional Array

Figure 8-3 shows how you map an entire horizontally-indexed scrolled array using the %ALL syntax. The %ALL syntax maps to and from both the arrays on the EMPLOYEE_DEPENDENT_FORM. Since you do not specify the subscript values, RDU determines the number of elements to map to both scrolled and horizontally-indexed scrolled arrays:

- In the case of the scrolled form array PARENT_LAST_NAME, RDU maps five elements. That is, it maps the number of elements in the record array PARENT_LAST_NAME.
- In the case of the two-dimensional, horizontally-indexed scrolled array CHILD_NAME:
    - RDU maps two elements to and from the indexed portion of the form. That is, it maps the number of elements in the second dimension of the record array CHILD_NAME. (Note that the second dimension of the form array has three elements while the second dimension of the record array has only two elements.)

RDU maps five elements to the scrolled area, the number of elements in the first dimension of the record array CHILD_NAME.



Two-Dimensional Group Record Array

```
DEFINE RECORD FAMILY_RECORD.
FAMILY STRUCTURE.
  FAMILY_NUMBER      DATATYPE NUMERIC  5 DIGITS.
  FAMILY_NAME STRUCTURE OCCURS 5 TIMES.          <--- Occurs five times
    PARENT_LAST_NAME DATATYPE TEXT    15.
    CHILD_NAME OCCURS 2 TIMES                    <--- Two-dimensional
                     DATATYPE TEXT     5.             array occurs
  END FAMILY_NAME STRUCTURE.                          two times
END FAMILY STRUCTURE.
END FAMILY_RECORD.
```

Request

```
FORM IS      EMPLOYEE_DEPENDENT_FORM;
RECORD IS    FAMILY_RECORD;

DISPLAY FORM EMPLOYEE_DEPENDENT_FORM;

  OUTPUT  %ALL;
  INPUT  %ALL;

END DEFINITION;
```

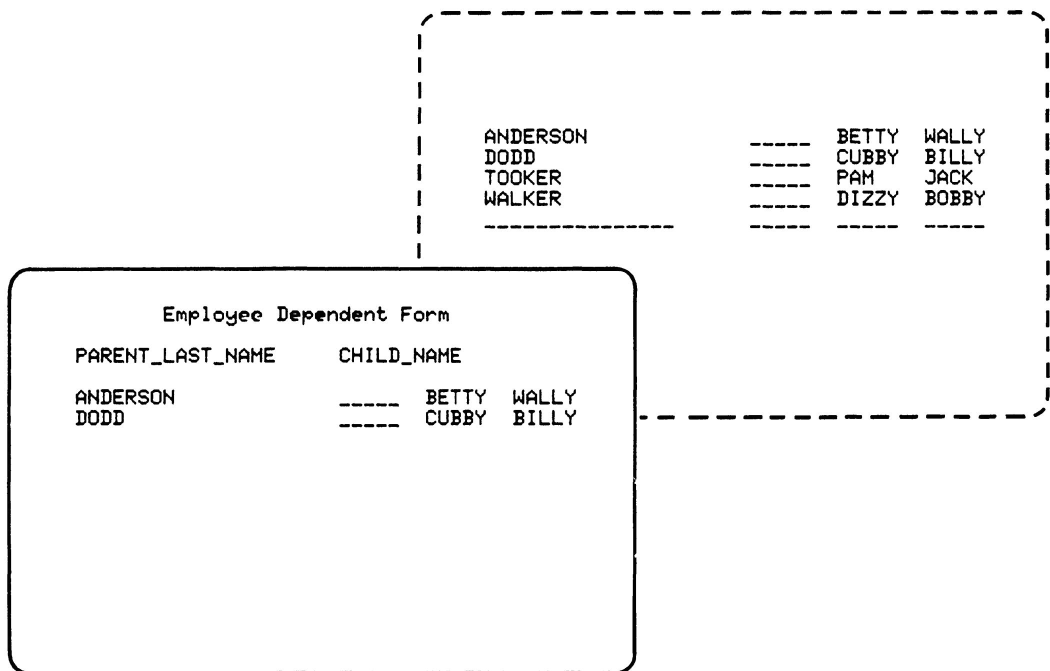
Figure 8-3: Using %ALL Syntax to Map Two-Dimensional Arrays

8.5.4 Mapping a Subset of a Two-Dimensional Array

Figure 8-4 shows how to map a subset of an entire form or record array by explicitly specifying the subscript values. It also shows that you can specify different subscripts in the sending array than in the receiving array as long as the number of elements you map is the same.

The request maps elements 2 to 5 of the one-dimensional record array PARENT_LAST_NAME to the first four lines of the scrolled form array PARENT_LAST_NAME. It also outputs elements 1 to 2 from the record array CHILD_NAME.

Note that the request maps these record array elements to a different subset of elements on the form. That is, the record elements 2 to 5 from the FAMILY_NAME group array are displayed in the form array on rows 1 to 4. The record elements 1 to 2 of the CHILD_NAME array are displayed on the horizontally-indexed fields 2 to 3.



(continued on next page)

Figure 8-4: Mapping a Subset of a Two-Dimensional Array

Two-Dimensional Group Record

```
DEFINE RECORD FAMILY_RECORD.
FAMILY STRUCTURE.
  FAMILY_NUMBER      DATATYPE NUMERIC  5 DIGITS.
  FAMILY_NAME STRUCTURE OCCURS 5 TIMES.
    PARENT_LAST_NAME DATATYPE TEXT    15.
    CHILD_NAME OCCURS 3 TIMES
                     DATATYPE TEXT     5.
  END FAMILY_NAME STRUCTURE.
END FAMILY STRUCTURE.
END FAMILY_RECORD.
```

← Group array occurs five times

← Two-dimensional array occurs three times

Request

```
FORM IS      EMPLOYEE_DEPENDENT_FORM;
RECORD IS    FAMILY_RECORD;

DISPLAY FORM EMPLOYEE_DEPENDENT_FORM;

OUTPUT
     PARENT_LAST_NAME[2 TO 5]
        TO PARENT_LAST_NAME[1 TO 4],
     CHILD_NAME[2 TO 5, 1 TO 2]
        TO CHILD_NAME[1 TO 4, 2 TO 3];
 WAIT;
END DEFINITION;
```

Figure 8-4: Mapping a Subset of a Two-Dimensional Array (Cont.)

Using an Array as a Control Value 9

Chapter 6, Using Conditional Instructions in Requests, described how to use conditional instructions to execute different request instructions depending on a control value. This chapter explains how to use conditional instructions to input or output selective elements in an array depending on a control value array. For instance, by using a conditional instruction, you can specify that TDMS collect only those elements in a scrolled field that an operator did not enter correctly the first time.

9.1 How to Use an Array As a Control Value to Collect Varying Elements

For TDMS to display or collect selective elements at run time, you can specify a conditional instruction that uses both:

- An array as a control value
- A variable as a subscript to specify and map elements that are in a form or record array

The array you specify as a control value must use a contiguous range of subscript values called a **dependent range**. For instance, in the following phrase there are two dependent ranges, 1 to 10 and 1 to 5:

```
CONTROL FIELD IS CONTROL_ARRAY[1 TO 10, 1 TO 5]
```

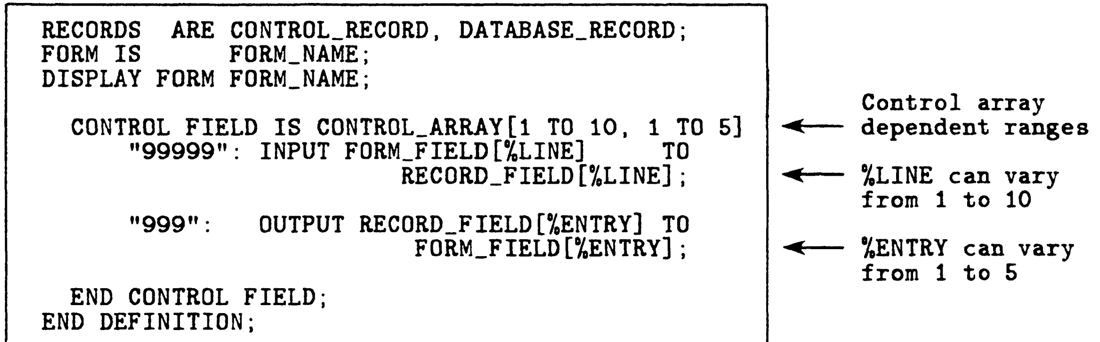
The variable subscript you specify can be one of two keywords: %LINE or %ENTRY. These subscript names are known as **dependent names**. You can use a dependent name as a subscript in mapping instructions:

```
OUTPUT RECORD_FIELD[%LINE] TO FORM_FIELD[%LINE];
```

The value of the dependent name varies at run time depending on the values in the dependent range. For instance, in Figure 9-1, %LINE and %ENTRY vary in value depending on the dependent ranges in the control value array CONTROL_ARRAY.

At run time:

- %LINE takes on the values 1 to 10
- %ENTRY takes on the values 1 to 5




```
RECORDS  ARE CONTROL_RECORD, DATABASE_RECORD;
FORM IS      FORM_NAME;
DISPLAY FORM FORM_NAME;

  CONTROL FIELD IS CONTROL_ARRAY[1 TO 10, 1 TO 5]
      "99999": INPUT FORM_FIELD[%LINE]     TO
                            RECORD_FIELD[%LINE];

      "999":   OUTPUT RECORD_FIELD[%ENTRY] TO
                             FORM_FIELD[%ENTRY];

  END CONTROL FIELD;
END DEFINITION;
```

Figure 9-1: Using an Array as a Control Value

Figure 9-1 shows that when you declare a control value array, RDU automatically assigns the subscript range values to the dependent names %LINE and %ENTRY. By default, RDU always assigns the first dimension in a control value array to %LINE and the second dimension to %ENTRY.

9.2 How TDMS Evaluates a Control Value Array at Run Time

When a program calls a request, TDMS evaluates the control value, element by element. As with any conditional instruction, TDMS checks that the value in the control value (in this case, the value in an element) matches the case value in the request. When TDMS finds a match, it executes the instructions associated with that case value. For a one-dimensional control array, %LINE has the same subscript value as the element TDMS is currently evaluating.

For instance, suppose you specify a CONTROL FIELD IS instruction that:

- Uses a one-dimensional array of five elements as a control value
- Has an associated conditional mapping instruction using the dependent name %LINE

CONTROL_ARRAY[1 TO 5] (in record CONTROL_RECORD)

99999	Element 1
	Element 2
	Element 3
99999	Element 4
99999	Element 5

Request

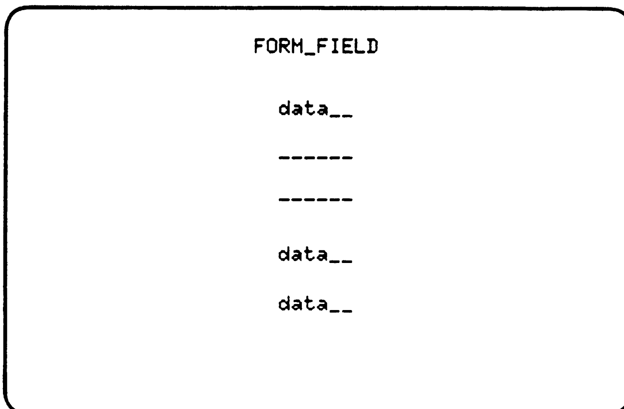
```
RECORDS ARE CONTROL_RECORD,
            DATABASE_RECORD;
FORM IS     FORM_NAME;
        .
        .
        .

   CONTROL FIELD IS CONTROL_ARRAY[1 TO 5]
       "99999": INPUT FORM_FIELD[%LINE] TO
                      RECORD_FIELD[%LINE];
   END CONTROL FIELD;

END DEFINITION;
```

Suppose also that, at run time, elements 1, 4, and 5 in the control value array CONTROL_ARRAY contain the value 99999. Figure 9-2 shows that when TDMS finds the value 99999 in CONTROL_ARRAY elements 1, 4, and 5, it executes the conditional mapping instruction specified in the request.

FORM_NAME



RECORD_FIELD
(in record
DATABASE_RECORD)

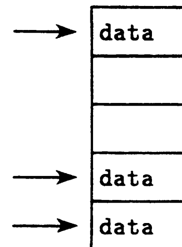


Figure 9-2: How TDMS Evaluates a Control Value Array at Run Time

The value of %LINE in the mapping instructions in Figure 9-2 is, successively, 1, 4, and 5. TDMS therefore inputs:

```
FORM_FIELD[1] TO  RECORD_FIELD[1]
FORM_FIELD[4] TO  RECORD_FIELD[4]
FORM_FIELD[5] TO  RECORD_FIELD[5]
```

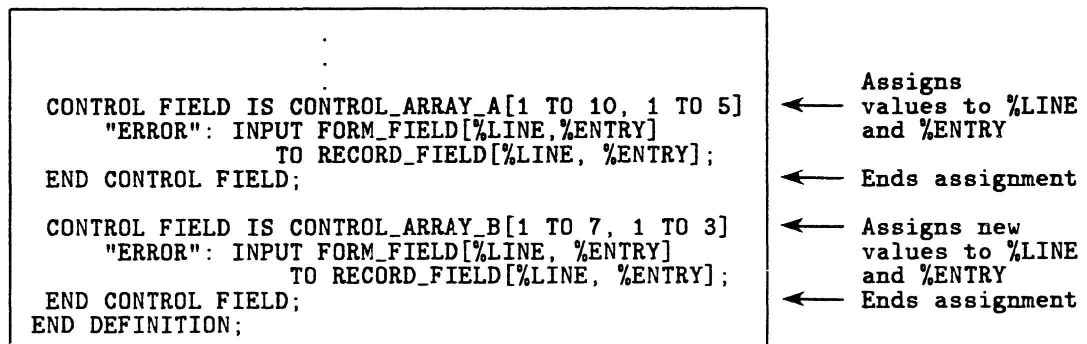
9.3 Rules for Specifying the Control Value Array

You must observe the following rules when using an array as a control value:

- The array must be TEXT data type.
- You must explicitly specify the dependent range or dependent ranges. You can specify the entire array or a subset of the array.
- You cannot nest CONTROL FIELD IS instructions that use dependent ranges.
- The array can be a one- or two-dimensional array.
- When you end a conditional instruction with the ending key phrase (END CONTROL FIELD), a dependent range is no longer active; that is, it is no longer assigned to a dependent name.

Once you specify a control value array range and RDU assigns that range to a TDMS-defined dependent name, the range remains associated with that dependent name until you end that control value with an END CONTROL FIELD phrase.

Figure 9-3 shows that when you end a CONTROL FIELD IS instruction with an END CONTROL FIELD phrase, a dependent range is no longer active. The two TDMS-defined dependent names can be reassigned in a new CONTROL FIELD IS instruction. They cannot be referred to until they are reassigned.



```
                         .
                         .
                         .
  CONTROL FIELD IS CONTROL_ARRAY_A[1 TO 10, 1 TO 5]     <--  Assigns values to %LINE and %ENTRY
      "ERROR": INPUT FORM_FIELD[%LINE,%ENTRY]
                  TO RECORD_FIELD[%LINE, %ENTRY];
  END CONTROL FIELD;                                    <--  Ends assignment

  CONTROL FIELD IS CONTROL_ARRAY_B[1 TO 7, 1 TO 3]      <--  Assigns new values to %LINE and %ENTRY
      "ERROR": INPUT FORM_FIELD[%LINE, %ENTRY]
                   TO RECORD_FIELD[%LINE, %ENTRY];
  END CONTROL FIELD;                                    <--  Ends assignment
 END DEFINITION;
```

Figure 9-3: The Scope of a Dependent Range

If you attempt to declare an additional dependent range before you end a current one (by nesting CONTROL FIELD IS instructions that use dependent ranges), RDU returns an error message, as Figure 9-4 shows.

```
                          .
                          .
                          .
CONTROL FIELD IS CONTROL_ARRAY[1 TO 10]            <-- Assigns values
   "ERROR": INPUT                                      to %LINE
            FORM_FIELD[%LINE] TO RECORD_FIELD[%LINE];
010         CONTROL FIELD IS NESTED_ARRAY_2[1 TO 10]   <-- INCORRECT
%RDU-E-ILLDEPRNG, Illegal nested dependent ranges          dependent
                                                           range
                "BAD": INPUT FORM_FIELD[%LINE] TO
                             RECORD_FIELD[%LINE];
            END CONTROL FIELD;
END CONTROL FIELD;
```

Figure 9-4: Illegal Nested Dependent Ranges

In addition, if you attempt to declare a three-dimensional array as a control value, RDU returns errors messages.

Figure 9-5 shows how you can specify one element of an array as a control value or a nonarray control value within a single conditional instruction. (%LINE and %ENTRY do not get assigned to single element or nonarray control values.)

```
                          .
                          .
                          .
                                                         Assigns
CONTROL FIELD IS ARRAY[1 TO 10]                      <-- values
  "999":                                                 to %LINE
     INPUT FORM_ARRAY[%LINE] TO RECORD_ARRAY[%LINE];
        CONTROL FIELD IS SIMPLE_FIELD
           "ERR": OUTPUT RECORD_FIELD TO FORM_FIELD;
                  CONTROL FIELD IS ONE_ELEMENT_ARRAY[3]
                     "NOT": MESSAGE LINE IS "WRONG";
                  END CONTROL FIELD;
        END CONTROL FIELD;
END CONTROL FIELD;                                   <-- Ends
END DEFINITION;                                          assignment
```

Figure 9-5: Specifying Nonarray or Single-Element Control Values

9.3.1 Explicitly Assigning Values to %LINE and %ENTRY

In some cases, you may wish to specify the assignment of %LINE and %ENTRY explicitly. You can include the dependent names followed by an equal sign in the CONTROL FIELD IS phrase:

```
CONTROL FIELD IS CONTROL_ARRAY[%ENTRY=1 TO 10, %LINE= 1 TO 5]
```

Figure 9-6 shows that you follow the dependent names by specific subscript values. This allows you to specify %ENTRY as the first dependent name and %LINE as the second dependent name.

```
                          .
                          .
                          .
   CONTROL FIELD IS
   CONTROL_ARRAY[%ENTRY= 1 TO 10, %LINE= 1 TO 5]
      "ERROR": INPUT FORM_FIELD[%ENTRY, %LINE] TO
                     RECORD_FIELD[ %ENTRY, %LINE];

   END CONTROL FIELD;
END DEFINITION;
```

Figure 9-6: Explicitly Assigning Values to %LINE and %ENTRY

9.3.2 Using a Work Array as a Control Value Array

You may want to use the array to which you input data from a scrolled or indexed form array as your control array. TDMS can then search the array for the case values you specify in a request.

More often, however, as with all control values, you may want to use a **work array** as a control value. A work array is separate from the array containing database information. The application program can analyze the data in your database array and place values in corresponding elements of a work array. By using a separate work array, you avoid the danger of mapping extraneous data to your database. The work array need not contain the same number of elements as the array to which you are mapping database information.

9.3.3 Specifying an Entire Array as a Control Value

If you use an entire array as a control value, you must specify 1 as the lower subscript. Since some record arrays may be created in the CDD with the beginning subscript of 0 or a minus number, as with all array records, RDU adjusts that array to be a one-based array. RDU generates a warning message indicating the new subscript values that it assigns to the record array:

```
CONTROL_ARRAY 0:9
%RDU-W-CHNGBND, changing the bounds of the array from 0:9 to 1:10
```

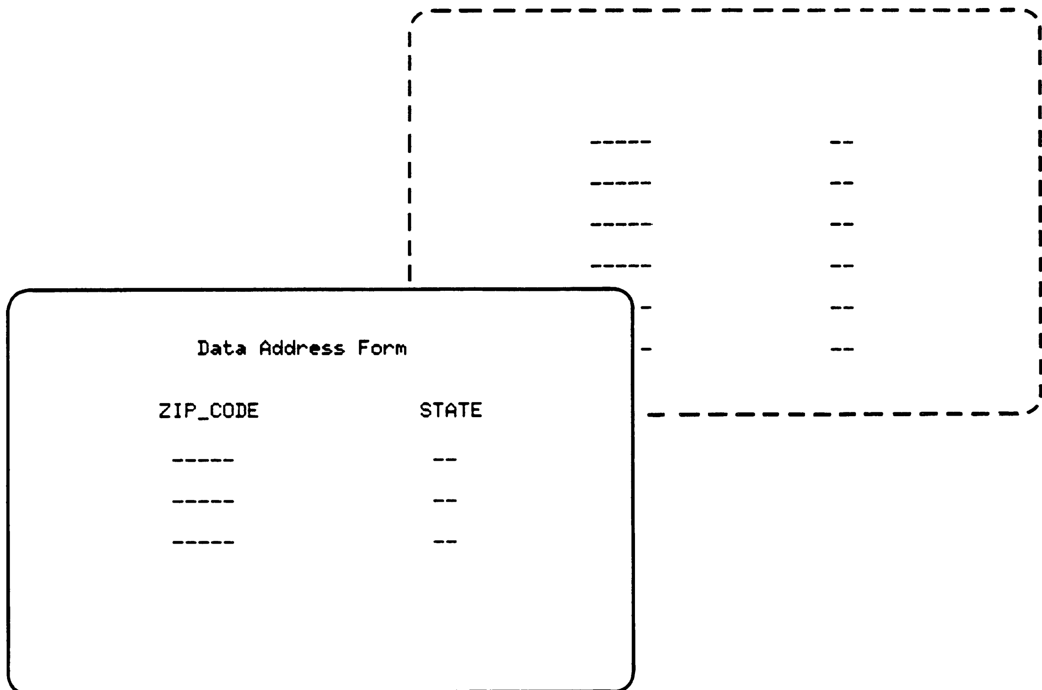
When you use this array as a control value, you must specify the elements 1 to 10:

```
CONTROL_ARRAY[1 TO 10]
```

9.4 Example - Using a One-Dimensional Control Value Array

Figure 9-7 shows a request using a one-dimensional control array. Mapping instructions using %LINE collect and display data in several scrolled form fields. To create this request, you must:

- Declare an array with a dependent range as a control value. In this example, the control array CONTROL_ZIP_STATE is the control value.
- Specify as a case value the error value that you want TDMS to check. In this example, TDMS tests each element in the control value array CONTROL_ZIP_STATE for the case value 99999.
- Specify the %LINE dependent name in the mapping instructions associated with the case value.



(continued on next page)

Figure 9-7: Collecting Elements from Several Scrolled Fields

Records

```
DEFINE RECORD DATA_ADDRESS_RECORD.
DATA_ADDRESS STRUCTURE.
 ADDRESS STRUCTURE OCCURS 6 TIMES.
   ZIP_CODE DATATYPE NUMERIC 5.
   STATE    DATATYPE TEXT    2.
 END ADDRESS STRUCTURE.
END DATA_ADDRESS STRUCTURE.
END DATA_ADDRESS_RECORD.
```

```
DEFINE RECORD ZIP_STATE_CONTROL_REC.
ZIP_STATE_CONTROL_REC STRUCTURE.
   START_PROGRAM
         DATATYPE TEXT 5.
   CONTROL_ZIP_STATE
      OCCURS 6 TIMES
         DATATYPE TEXT 5.
END ZIP_STATE_CONTROL_REC STRUCTURE.
END ZIP_STATE_CONTROL_REC.
```

Request

```
FORM IS   DATA_ADDRESS_FORM;
RECORD IS DATA_ADDRESS_RECORD;
RECORD IS ZIP_STATE_CONTROL_REC;

DISPLAY FORM DATA_ADDRESS_FORM;

    CONTROL FIELD IS START_PROGRAM
        "BEGIN":
            INPUT %ALL;                                            <--- Collects
            DESCRIPTION /* Input ZIP_CODE[1 to 6]                       initial
                           and  STATE[1 TO 6] */;                       zip code
    END CONTROL FIELD;                                                  and state
                                                                        data
    CONTROL FIELD IS CONTROL_ZIP_STATE[1 TO 6]                 <--- Control
        "99999":                                                        array
            BOLD FIELD ZIP_CODE[%LINE], STATE[%LINE];          <--- Bolds and
            BLINK FIELD ZIP_CODE[%LINE], STATE[%LINE];                  blinks
                                                                        incorrect
                                                                        fields

            OUTPUT STATE[%LINE]     TO STATE[%LINE],           <--- Outputs
                   ZIP_CODE[%LINE]  TO ZIP_CODE[%LINE];                 incorrect
                                                                        data
            MESSAGE LINE IS
"Zip code and state information conflict. Please reenter";

            INPUT ZIP_CODE[%LINE]   TO ZIP_CODE[%LINE],        <--- Collects
                  STATE[%LINE]      TO STATE[%LINE];                    new data
  END CONTROL FIELD;
END DEFINITION;
```

Figure 9-7: Collecting Elements from Several Scrolled Fields (Cont.)

When an application program calls the request in Figure 9-7, the following sequence of events occurs:

1. When the program first calls the request, TDMS evaluates both control values. (The program places "BEGIN" in the first control value and clears the control value array before it calls the request.)

2. TDMS executes the instructions following the "BEGIN" case value in the first control value. It collects all the elements from the form field ZIP_CODE[1 TO 6] and STATE[1 TO 6] and returns them to record arrays ZIP_CODE[1 TO 6] and STATE[1 TO 6].

3. The program checks that each zip code is associated with an appropriate state abbreviation.

4. When there is an error in any element of the arrays ZIP_CODE or STATE, the program places a 99999 in the corresponding array element of the control value array, CONTROL_ZIP_STATE[1 to 6]. (It also clears the control value START_PROGRAM.)

   Suppose the application program finds that information in elements 2, 4, and 6 of the arrays ZIP_CODE and STATE is not compatible. The program places a 99999 in the corresponding elements 2, 4, and 6 of the control value array, CONTROL_ZIP_STATE. For example:

   CONTROL_ZIP_STATE

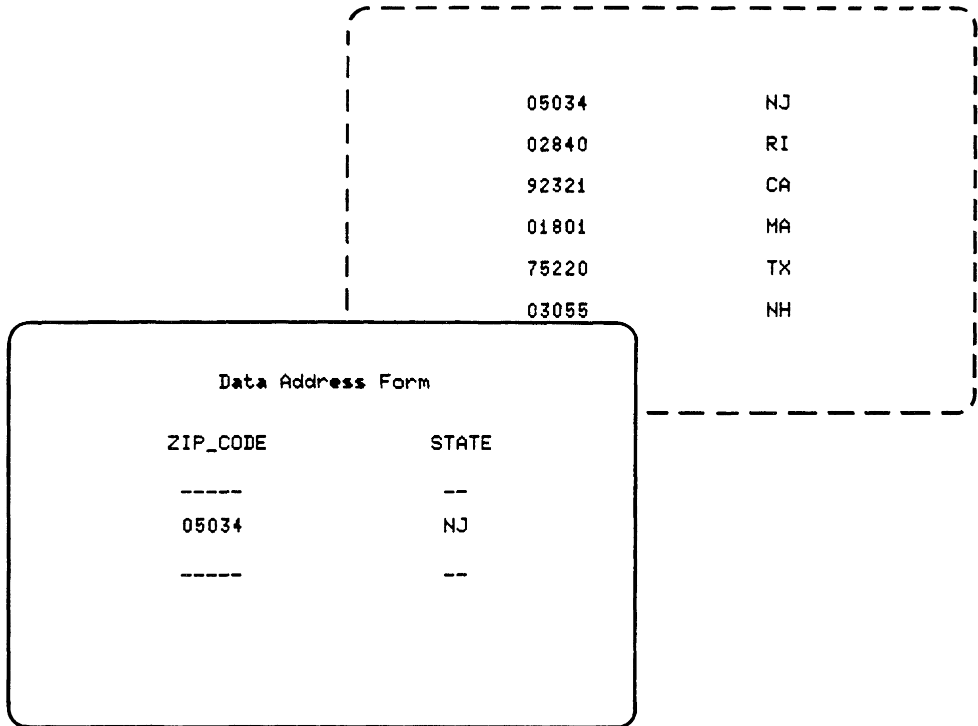
	Element 1
99999	Element 2
	Element 3
99999	Element 4
	Element 5
99999	Element 6

5. The second time the request is called, TDMS evaluates the control value array, element by element. It executes the mapping instructions following the matching 99999 case value in the request. TDMS:

   - Bolds and blinks the fields ZIP_CODE[2] and STATE[2], ZIP_CODE[4] and STATE[4], ZIP_CODE[6] and STATE[6]

   - Displays the incorrect data in those same fields

- Displays the message from the MESSAGE LINE IS instruction
- Allows the operator to input new zip code and state data in those same fields



9.5 Example - Using a Two-Dimensional Control Value Array

TDMS allows you to collect selected elements from a horizontally-indexed scrolled field. To do so, you can use a two-dimensional array as a control value. In Figure 9-8, %LINE takes on the subscript values of the first dependent range. %ENTRY takes on the subscript values of the second dependent range.

The request maps between:

- Two scrolled form arrays PROJECT_NUMBERS and TOTAL_EMPLOYEES and two subfields in the group record array PROJECT
- The horizontally-indexed scrolled form array WAGE_CLASS and the subfield WAGE_CLASS within the two-dimensional record array WAGE_CLASSES

PROJECT_SUMMARY_FORM

```
              Project Summary Form


   PROJECT    TOTAL          WAGE CLASS
   NUMBERS  EMPLOYEES    FIRST   SECOND    THIRD



   -----      ---         ---     ---       ---

   -----      ---         ---     ---       ---

   -----      ---         ---     ---       ---
```

Record

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY STRUCTURE.
  PROJECT STRUCTURE OCCURS 10 TIMES.
    PROJECT_NUMBERS DATATYPE TEXT 5.
    TOTAL_EMPLOYEES DATATYPE NUMERIC 3.
    WAGE_CLASSES  STRUCTURE OCCURS 3 TIMES.      <-- Two-dimensional array
       WAGE_CLASS DATATYPE NUMERIC 3.
    END WAGE_CLASSES STRUCTURE.
  END PROJECT STRUCTURE.
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_SUMMARY_RECORD.
```

Record

```
DEFINE RECORD PROJECT_SUMMARY_CONTROL_RECORD.
PROJECT_SUMMARY_CONTROL STRUCTURE.
  START_PROGRAM DATATYPE TEXT 5.
  CONTROL_PROJECT ARRAY 1:10,1:3  DATATYPE TEXT 3.      <-- Two-dimensional array
END PROJECT_SUMMARY_CONTROL STRUCTURE.
END PROJECT_SUMMARY_CONTROL_RECORD.
```

Request

```
FORM IS        PROJECT_SUMMARY_FORM;
RECORD IS      PROJECT_SUMMARY_RECORD;
RECORD IS      PROJECT_SUMMARY_CONTROL_RECORD;

 DISPLAY FORM PROJECT_SUMMARY_FORM;
   CONTROL FIELD IS START_PROGRAM
      "BEGIN":
         INPUT
           PROJECT_NUMBERS[1 TO 10]
                   TO PROJECT_NUMBERS[1 TO 10],
```

(continued on next page)

Figure 9-8: Using a Two-Dimensional Array as a Control Value

```
         WAGE_CLASS[1 TO 10, 1 TO 3]
                 TO WAGE_CLASS[1 TO 10,1 TO 3];
   END CONTROL FIELD;

   CONTROL FIELD IS CONTROL_PROJECT[1 TO 10,1 TO 3]
      "999":
        OUTPUT
         PROJECT_NUMBERS[%LINE]
                 TO PROJECT_NUMBERS[%LINE],
         TOTAL_EMPLOYEES[%LINE]
                 TO TOTAL_EMPLOYEES[%LINE],
         WAGE_CLASS[%LINE,1 TO 3]
                 TO WAGE_CLASS[%LINE, 1 TO 3];

       BOLD FIELD  TOTAL_EMPLOYEES[%LINE],
                   WAGE_CLASS[%LINE,%ENTRY];
       BLINK FIELD TOTAL_EMPLOYEES[%LINE],
                   WAGE_CLASS[%LINE, %ENTRY];
       INPUT
         WAGE_CLASS[%LINE,%ENTRY]
                 TO WAGE_CLASS[%LINE, %ENTRY];

      MESSAGE LINE IS
"Wage class totals are incorrect. Please reenter";

   END CONTROL FIELD;
END DEFINITION;
```

Figure 9-8: Using a Two-Dimensional Array as a Control Value (Cont.)

At run time:

1. When the program first calls the request in Figure 9-8, TDMS collects all the elements from the form field PROJECT_NUMBERS[1 TO 10] and WAGE_CLASS[1 TO 10, 1 to 3] and returns them to matching record arrays. (Before calling the request, the program places "BEGIN" in the control value START_PROGRAM and clears the control value array CONTROL_PROJECT.)

2. The program checks that for each project number, the number of employees specified in each wage class does not exceed thirty percent in wage classes 1 and 2 and forty percent in wage class 3.

3. When there is an error in any one of the wage class numbers, the program places a 999 in the corresponding array element of the control value array, CONTROL_PROJECT[1 to 10, 1 to 3]. (It also clears the control value START_PROGRAM.) For instance, if the program finds errors in the data returned to WAGE_CLASS[1,3], WAGE_CLASS[3,2], and WAGE_CLASS[9,1], it places a 999 in the corresponding elements of the control array as in the following example.

CONTROL_PROJECT Array

Column 1	Column 2	Column 3	
		999	Row 1
			Row 2
	999		Row 3
	. . .		
			Row 8
999			Row 9
			Row 10

4. The second time the request is called, TDMS evaluates the control value array, element by element. When it finds a matching 999 in an element of the control value array, it executes the mapping instructions following the case value. TDMS:

   - Bolds and blinks the element of the form field WAGE_CLASS in which the error occurs and associated element of the field TOTAL_EMPLOYEES.
   - Displays a message indicating the wage class totals are incorrect.
   - Displays the information related to that incorrect data. The project number, the total number of employees assigned to that project, the incorrect wage class number, and the correct numbers in the other two wage classes are displayed. This lets the operator see the correct data associated with the error.
   - Allows the operator to enter a correct number in the wage class field in error, WAGE_CLASS[1,3], WAGE_CLASS[3,2], WAGE_CLASS[9,1].

When the request ends, the application program recalculates the percentages. If there are still errors, TDMS collects further wage class information or goes back to collecting new project and wage class information. For example:

```
67854   _12   __3   __3   __6
32412   _42   _21   _12   _32
-----   ---   ---   ---   ---
-----   ---   ---   ---   ---
-----   ---   ---   ---   ---
              ---   ---   ---
              _20   _38   __6
              ---   ---   ---
```

```
                 Project Summary Form

PROJECT    TOTAL            WAGE CLASS
NUMBERS  EMPLOYEES     FIRST   SECOND    THIRD


67854      _12          __3     __3       __6

-----      ---          ---     ---       ---

32412      _42          _21     _12       _32
```

Note that you can combine the use of a dependent name with a specific range as did the following mapping instruction in the request in Figure 9-8:

```
OUTPUT WAGE_CLASS[%LINE, 1 TO 3] TO WAGE_CLASS[%LINE, 1 TO 3];
```

How to Display and Input Data in a Scrolled Region 10

Whether a specific element of a record array is displayed in the form window or whether an element of a form array can be accessed to enter new data, depends on several conditions, including:

- Operator action at run time
- Kinds of mappings that you create

To view the data in a scrolled form array, the operator must use the TAB, the BACK SPACE, the up arrow, and the down arrow keys on the keyboard to move from input field to input field:

- The TAB key moves the cursor from one input field to the next.
- The BACK SPACE key moves the cursor from one input field back to the immediately previous input field.
- The up arrow key scrolls data in the window back up toward the top of the array; for example, from element[15] back to element[3].
- The down arrow key scrolls data down toward the bottom of the array; for example, from element[1] to element[15].

The elements from the underlying virtual array scroll through the form array window. This chapter describes how the request can control the display of a scrolled region.

Because TDMS moves the cursor from one field mapped for input to the next field mapped for input, you must map at least one field on each line of the scrolled form region for input if you want the operator to be able to tab to each line in a scrolled form region.

10.1 How TDMS Displays and Collects Data in a Scrolled Array

The following example shows how an operator accesses elements (fields) that are mapped for input in a scrolled array. The form contains a single scrolled region containing two scrolled arrays, UNIVERSITY and DEGREE. The visible window is five lines long. The request shows the specific mappings between the form and record that you wish to create. Although this may not be a typical mapping, it illustrates how TDMS allows the operator to move through a scrolled array and access the fields you map for input.

EMPLOYEE_ADD_EDUCATION

```
              Employee Add Education


       UNIVERSITY:                   DEGREE:

         1___________________           _____

         2___________________           _____

         3___________________           _____

         4___________________           _____

         5___________________           _____
```

EDUC_RECORD

```
DEFINE RECORD EDUC_RECORD.
 EDUC_RECORD STRUCTURE.
   EDUC_EMPL_NUMBER  DATATYPE UNSIGNED LONGWORD.
   EDUC_EMPL STRUCTURE OCCURS 15 TIMES.
     EDUC_UNIVERSITY DATATYPE TEXT 20.
     EDUC_DEGREE     DATATYPE TEXT  5.
   END EDUC_EMPL STRUCTURE.
 END EDUC_RECORD STRUCTURE.
END EDUC_RECORD.
```

EDUC_REQUEST

```
FORM IS       EMPLOYEE_ADD_EDUCATION;
RECORD IS     EDUC_RECORD;

 DISPLAY FORM EMPLOYEE_ADD_EDUCATION;

 INPUT UNIVERSITY[1]  TO EDUC_UNIVERSITY[1],
       UNIVERSITY[3]  TO EDUC_UNIVERSITY[3],
       UNIVERSITY[7]  TO EDUC_UNIVERSITY[7],
       UNIVERSITY[15] TO EDUC_UNIVERSITY[15];
```

```
  INPUT DEGREE[1]       TO  EDUC_DEGREE[1],
        DEGREE[3]       TO  EDUC_DEGREE[3],
        DEGREE[7]       TO  EDUC_DEGREE[7],
        DEGREE[15]      TO  EDUC_DEGREE[15];
END DEFINITION;
```

In the preceding example, the lines in the scrolled region of the form are numbered for ease of explanation. These numbers do not actually appear on the screen.

When TDMS executes the request instructions:

1. The form is displayed on the screen. The operator sees five lines of the scrolled region. The operator can enter data on those lines that you map for input (up to the total number of elements in the receiving record array).
2. The cursor is automatically positioned at the first character of the first field mapped for input, in this case UNIVERSITY[1]. Note that the cursor moves only to those fields mapped for input.
3. When the operator has filled the UNIVERSITY[1] field and presses the TAB key, the cursor moves horizontally to the DEGREE[1] field on the same line since it is also mapped for input.
4. After the operator fills the two input fields on the same line of the scrolled region and presses a TAB key, the cursor moves to the third row, since UNIVERSITY[3] is the next input field. No scrolling occurs. The cursor simply moves to the third line. Both the first and third lines remain in the window.
5. The operator enters data in form fields UNIVERSITY[3], presses the TAB key, and enters data in DEGREE[3].

```
             Employee Add Education


   UNIVERSITY:                    DEGREE:

     1___________________          _____
     2___________________          _____
     3 __________________          _____
     4___________________          _____
     5___________________          _____
```

6. The next field mapped for input is UNIVERSITY[7].

7. The operator presses the TAB key and the cursor moves to the next input field, UNIVERSITY[7].

8. Scrolling occurs because there cannot be seven lines in the window at one time. UNIVERSITY[7] is on a line that can be displayed in the window if TDMS scrolls the third line to the first position in the scrolled region. TDMS scrolls through the underlying virtual array until UNIVERSITY[7] is in the bottom row of the window. Note that TDMS always attempts to keep both fields in the window:

```
                Employee Add Education


     UNIVERSITY:                    DEGREE:
      3__________________             _____
      4__________________             _____
      5__________________             _____
      6__________________             _____
      7 _________________             _____
```

9. The operator presses the TAB key after entering data in DEGREE[7].

10. The next field mapped for input is UNIVERSITY[15].

11. TDMS cannot display both fields in the window because UNIVERSITY[15] is more than four lines away from the last field mapped for input, DEGREE[7]. TDMS "jumps" to the next field mapped for input.

12. TDMS places the new input field, UNIVERSITY[15], in the last line of the scrolled form field. Line 15 is in the bottom position in the window. Intervening lines (lines 11 through 14) are seen by the operator.

```
              Employee Add Education


UNIVERSITY:                        DEGREE:

i1___________________                -----
i2___________________                -----
i3___________________                -----
i4___________________                -----
i5 __________________                -----
```

10.2 How to Allow the Operator to View Data in Scrolled Regions

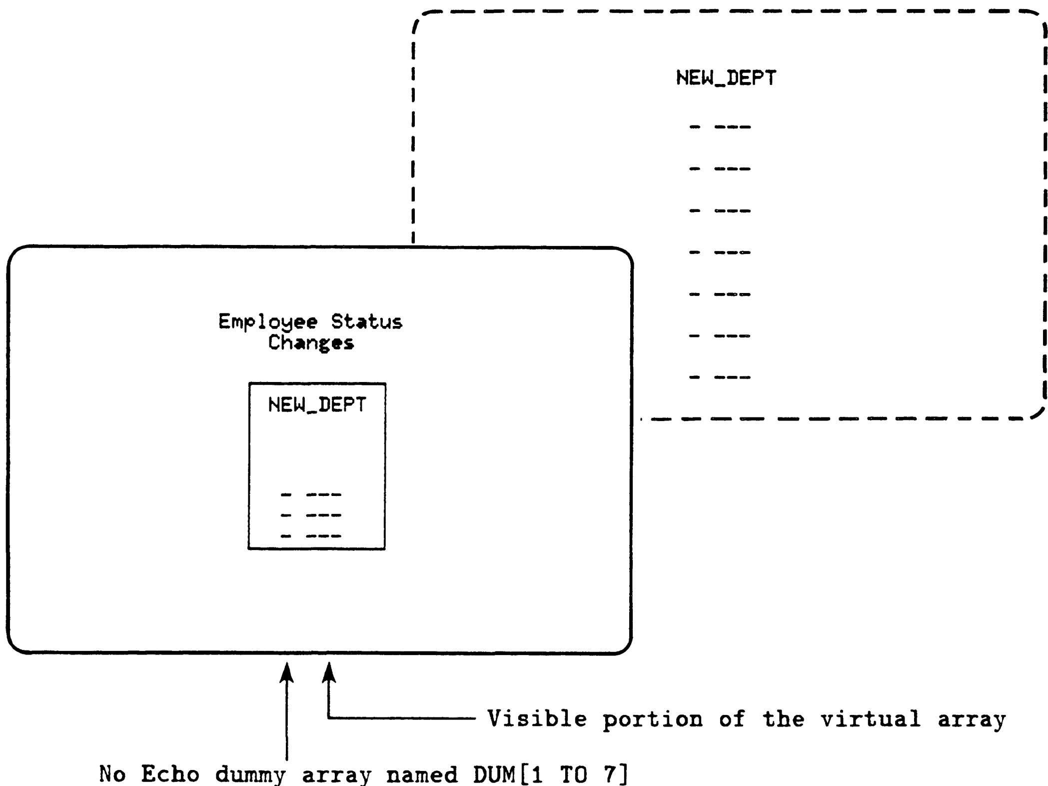
TDMS moves through a scrolled region from input field to input field. If you create a request that displays data to a scrolled array but contains no input mappings for that scrolled array, the operator cannot tab through the field to view all the data.

If you want to display a large record array in a scrolled form array but do not want to allow operator input in these fields, you can use the following technique:

- In your form definition, define a dummy, one-character form array. Each field of this dummy array appears at the beginning of each line of the scrolled form array. The dummy field should have the No Echo attribute. That is, anything the operator types in this field is not echoed back on the form screen.
- In your record definition, create a receiving array.
- In your request, define an input mapping from this dummy form array to the record array.
- At run time, the operator can then scroll through this form array, moving from dummy input field to dummy input field.

Note that the operator need not enter data in the dummy field. The array provides a mechanism to traverse the scrolled array. If the operator does enter data in a dummy field, it is discarded by the application program.

Figure 10-1 shows the request, the form, and the record definitions for an application that uses a dummy array.



Group Array

```
DEFINE RECORD HIST_RECORD.
HIST_CHANGES STRUCTURE
           OCCURS 7 TIMES.
  HIST_DEPT  DATATYPE TEXT 3.
  DUM        DATATYPE TEXT 1.
END HIST_CHANGES STRUCTURE.
END HIST_RECORD.
```

Request

```
FORM IS EMPLOYEE_STATUS_CHANGES;
RECORD IS HIST_RECORD;

 DISPLAY FORM EMPLOYEE_STATUS_CHANGES;

 INPUT  DUM[1 TO 7]       TO DUM[1 TO 7];
 OUTPUT HIST_DEPT[1 TO 7] TO NEW_DEPT[1 TO 7];

END DEFINITION;
```

Figure 10-1: Displaying Data in a Scrolled Region

At run time, when the request in Figure 10-1 is called, the following action occurs:

1. TDMS displays the seven element record array, HIST_CHANGES, subfield HIST_DEPT, in the form array NEW_DEPT.
2. The operator can tab through the array from form field DUM[1] to form field DUM[7], placing the cursor on each scrolled line. This allows the operator to view all the data that is output to that scrolled region, line by line.

Note that if the dummy array DUM were not present on the form and you did not map it for input, the operator would be unable to tab through the scrolled array.

Program Request Keys 11

This chapter discusses program request keys, how they work, and the correct syntax for using them. Several examples illustrate ways you can use program request keys in your TDMS applications.

Program request keys provide a convenient and efficient way for the operator to communicate with a TDMS application program.

By default, TDMS defines a number of keys to have a special function when the operator presses them at run time. These keys are shown in Table 11-1. You can use the PROGRAM KEY IS instruction to define other keys to supplement these. You cannot redefine any of the keys that already have special functions assigned by TDMS.

The description of the PROGRAM KEY IS instruction in the *VAX TDMS Reference Manual* lists the keys that can be redefined.

Table 11-1: Run-Time Function Keys

Key	Function
TAB	Moves the cursor to the next field on a form without changing the contents of the current field.
BACK SPACE (F12)	Moves the cursor to the previous field on a form without changing the contents of the current field.
LINE FEED (F13)	Deletes the contents of the field in which the cursor is positioned.

(continued on next page)

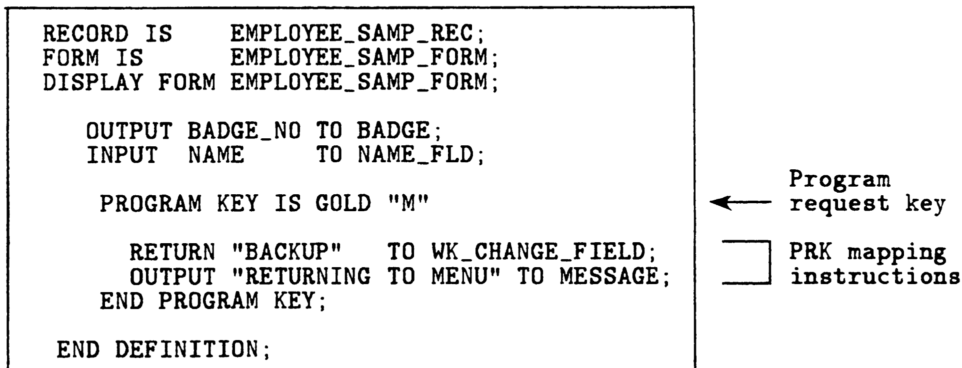
Table 11-1: Run-Time Function Keys (Cont.)

Key	Function
HELP (PF2 or F15)	Accesses help information and help forms. If the cursor is located at a field that has a help message, the help message is displayed the first time the operator presses the HELP key and the Help form is displayed the second time. Subsequent Help forms are displayed each time the operator presses HELP.
CTRL/R	Refreshes the screen.
CTRL/W	Refreshes the screen.
Right arrow	Moves to the right within a field.
Left arrow	Moves to the left within a field.
Down arrow	Moves down one line in a scrolled region.
Up arrow	Moves up one line in a scrolled region.
GOLD-down arrow	Exits from a scrolled region and moves the cursor to the next field.
GOLD-up arrow	Exits from a scrolled region and moves the cursor to the preceding field.
RETURN	Moves to the next form.
ENTER	Performs the same function as RETURN.
PF4	Prints an image of the active form if TSS$HARDCOPY is defined as one or more devices or file specifications.

11.1 What Are Program Request Keys?

Program request keys (PRKs) are keyboard and keypad keys that you can define in a request to have special meaning.

You define a PRK in a request by naming the key and associating mapping instructions with that key. When the TDMS application is running, if an operator presses a PRK that you defined, TDMS executes the mapping instructions you associated with that PRK and terminates the request. For instance, you can create the following request containing a program request key and associated PRK mapping instructions.



```
PRK_SAMPLE_REQUEST

RECORD IS    EMPLOYEE_SAMP_REC;
FORM IS      EMPLOYEE_SAMP_FORM;
DISPLAY FORM EMPLOYEE_SAMP_FORM;

   OUTPUT BADGE_NO TO BADGE;
   INPUT  NAME     TO NAME_FLD;

    PROGRAM KEY IS GOLD "M"

      RETURN "BACKUP"   TO WK_CHANGE_FIELD;
      OUTPUT "RETURNING TO MENU" TO MESSAGE;
    END PROGRAM KEY;

 END DEFINITION;
```


Figure 11-1: A Sample PRK Instruction

During a TDMS application, when an operator presses the PRK as defined in the PRK_SAMPLE_REQUEST (the GOLD-M key sequence on the keyboard), TDMS:

1. Executes the output and return mapping instructions associated with the PRK as follows:

   - Displays the string "RETURNING TO MENU" in the form field MESSAGE

   - Returns the text string "BACKUP" to the record field WK_CHANGE_FIELD

2. Terminates the request call and returns control to the application program

3. Returns a value to the program indicating that the request was terminated by a PRK

Note that when an operator presses a PRK at run time, TDMS checks that all form fields defined as Response Required have data entered in them (in the default mode of the PRK instruction). If these fields do not have data entered in them, TDMS ignores the PRK and continues executing the instructions in the request. (See the section on the default CHECK modifier and the NO CHECK modifier.)

11.2 Using Program Request Keys

PRKs are a convenient way for a TDMS operator to communicate with the application program.

By returning messages (that you predefine in a request) to the program, PRKs permit the operator to send run-time messages to the application program. The program can then respond to the condition identified by the operator.

You can use program request keys in your request to let the operator:

- Select a menu option. (For example, you might have a menu in which the operator selects an option by pressing a particular PRK.)
- Notify the application of a change in the sequence of operations. (For example, the program may continue asking for employee data until the operator presses a PRK to indicate readiness for a new employee form.)
- Notify a TDMS application program to exit.
- Indicate that a particular type of operator error occurred and that the program should take certain corrective action.

You can also use PRKs in conditional instructions. Later in this chapter, you see examples of program request key instructions that are used within a conditional instruction to return values to control values. First, however, you learn how to create a simple request containing a program request key.

11.3 Creating a Request That Uses a Program Request Key

The request in Figure 11-2 shows that to use a program request key in a request, you must specify:

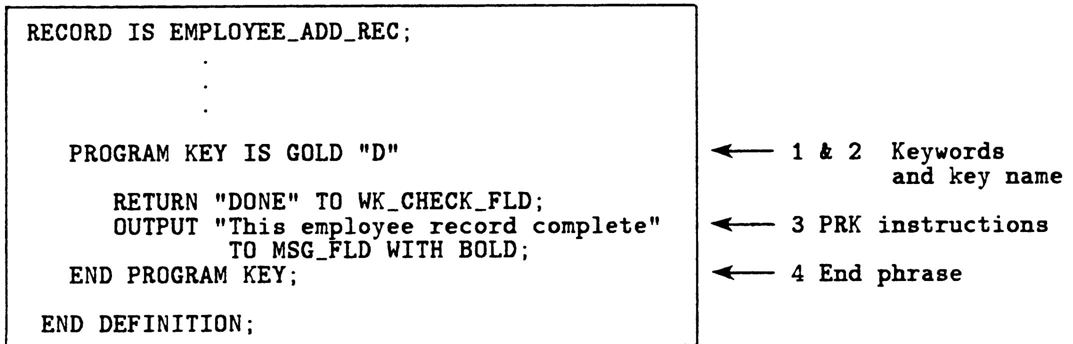
1. The instruction words, PROGRAM KEY IS
2. A PRK name, either:
   - The keyword GOLD followed by the name of any key valid with GOLD. (See the PROGRAM KEY IS instruction in the *VAX TDMS Reference Manual* for a list of keys that you can use as PRKs.)
   - The keyword KEYPAD and one key from the keypad: 0-9, hyphen (-), period (.), or comma (,).

The keypad must be set to Application mode using the KEYPAD MODE IS APPLICATION instruction. See the KEYPAD MODE IS instruction in the *VAX TDMS Reference Manual.*

3. The following instructions:

   - One RETURN quoted-string to record-field instruction
   - One of either (but not both):
     - A MESSAGE LINE IS instruction
     - An OUTPUT quoted-string instruction

4. The end phrase END PROGRAM KEY followed by a semicolon



```
RECORD IS EMPLOYEE_ADD_REC;
             .
             .
             .

   PROGRAM KEY IS GOLD "D"                                   <--- 1 & 2  Keywords
                                                                         and key name
      RETURN "DONE" TO WK_CHECK_FLD;
      OUTPUT "This employee record complete"                 <--- 3 PRK instructions
             TO MSG_FLD WITH BOLD;
   END PROGRAM KEY;                                          <--- 4 End phrase

 END DEFINITION;
```

Figure 11-2: Defining Program Request Keys

Notice that you must:

- Specify the keyword GOLD or KEYPAD
- Enclose the key name in quotation marks
- End the PROGRAM KEY IS instruction with the end phrase END PROGRAM KEY and a semicolon

Note also that you do not use a semicolon following the PROGRAM KEY IS prk-key-name phrase.

RDU automatically puts all PRK names in uppercase. So, for example, saying GOLD "A" is the same as GOLD "a". At run time, when the operator presses a PRK key, whether it is uppercase or lowercase, TDMS matches it to the PRK name in the request.

The PROGRAM KEY IS instruction can occur anywhere within a request (except in the header section). TDMS responds to program request keys only after it has executed all output mappings.

11.3.1 Default CHECK Mode Modifier

When you define a program request key, it has the default modifier, CHECK. (You can assign the NO CHECK modifier to the PRK instruction.) With checking in effect, at run time when an operator presses a PRK:

1. TDMS checks to see that all form fields defined as Response Required (that are also mapped for input) contain data
2. TDMS checks to see that all form fields defined as Must Fill fields are filled
3. TDMS checks to see that all field validators (Choice List, Range List, and Check Digits) are met
4. If conditions 1, 2, and 3 are met, TDMS:
    - Executes the PRK instructions and terminates the request
    - Returns data from all the form fields that were mapped for input or return
5. If the Response Required fields do not have data in them, the Must Fill fields are not filled, or the field validators are not met, TDMS ignores the PRK

Note that fields on a form that are mapped for input are not necessarily also defined as Response Required fields. When a PRK is pressed, therefore, TDMS may terminate a request even though the operator may not have entered data in all fields mapped for input.

The data returned to a record, therefore, may be any of the following:

- Data entered by the operator during the current call to the request
- Data output to the form fields during the current call to the request
- Data in the form fields from the immediately previous call to the request
- Data associated with the form fields by form definition defaults (if no other data is in the fields)

Figure 11-3 shows how the CHECK modifier works. At run time, when the operator presses the PRK, TDMS checks if the field BADGE has data entered into it by the operator.

```
NAME______________

BADGE  _3456656

DEPARTMENT   __

MSG_FLD
  CANCEL_OPERATION
```

← Response Required field (TDMS checks that data is in this field)

DEPT_INFO_REQUEST

```
FORM IS       DEPT_INFO_FORM;
RECORD IS     DEPT_INFO_RECORD;
DISPLAY FORM DEPT_INFO_FORM;

  INPUT NAME        TO NAME,
        BADGE       TO BADGE,
        DEPARTMENT  TO DEPT;

     PROGRAM KEY IS GOLD "C"
                  CHECK;
        RETURN "CANCEL"
                TO  MSG_FLD;
        OUTPUT "CANCEL OPERATION"
                TO  MSG_FLD;
     END PROGRAM KEY;

 END DEFINITION;
```

← Operator presses the GOLD-C key sequence at run time

Figure 11-3: Using the CHECK Modifier

If the field does have data entered in it, TDMS:

1. Outputs the string "CANCEL OPERATION" to the form field MSG_FLD
2. Returns the string "CANCEL" to the program
3. Terminates the request and returns the badge number entered by the operator
4. Returns the values that happen to be in the form fields NAME and DEPARTMENT (the operator did not enter values in these fields)
5. Returns a value to the program indicating that the request was terminated by a PRK and that Response Required fields were checked for input

If the field BADGE does not have data entered in it, TDMS:

1. Issues an error message indicating that BADGE is a Response Required field
2. Ignores the PRK and the associated PRK instructions
3. Executes the remaining input instructions in the request and terminates the request normally

11.3.2 NO CHECK Modifier

If you assign a NO CHECK modifier, at run time TDMS executes only the PRK instructions and terminates the request. That is, when TDMS terminates the request in the NO CHECK mode, it executes only the RETURN, OUTPUT, or MESSAGE LINE IS instructions within the PRK instruction. It does not:

- Check for Response Required, Must Fill fields, or field validators
- Execute any instructions outside the PRK instruction
- Execute any INPUT TO instructions

11.4 Examples of Using Program Request Keys

The following sections contain two examples of PRKs in requests.

11.4.1 Using PRKs to Allow the Operator to Control Application Flow

By defining PRKs that return strings to the program, you give the operator some control over the flow of the application.

The request in Figure 11-4 illustrates this concept.

This request contains a series of four program request keys. When the operator presses them, TDMS returns a string to signal the program to take one of four actions:

String	Action
"BACK"	Go back to a form displayed earlier in an application and redisplay it with information previously entered on that form. Save the information collected so far on this current form.
"SKIP"	Discard changes entered on this current form and go to the Menu form in this application so the operator can select a new form.
"DONE"	Save the data entered so far on this form and write it to the appropriate record. Go to the Menu form for another selection.

String	Action
"EXIT"	Exit this application and write all data collected to the appropriate records.

By using these program keys, the operator can:

1. Move among many forms or menu selections in an application (PRK keypad 7 in Figure 11-4).
2. Skip the form that appears on the screen and discard any information entered on this form during this call to the request (PRK keypad 8 in Figure 11-4).
3. Enter information on a form and indicate when he or she is done and wants information entered to be written to a file (PRK keypad 9 in Figure 11-4).
4. Exit the application program (PRK keypad 4 in Figure 11-4).

```
CHANGE_EDUCATION_REQUEST

    FORM   IS CHANGE_EDUCATION_FORM;
    RECORD IS EDUC_RECORD;
    RECORD IS CHANGE_WORKSPACE;

    CLEAR SCREEN;
    DISPLAY FORM CHANGE_EDUCATION_FORM;

       KEYPAD MODE IS APPLICATION;
          PROGRAM KEY IS KEYPAD "7"
            CHECK;
            RETURN "BACK" TO WK_CONTROL_FIELD;
          END PROGRAM KEY;

          PROGRAM KEY IS KEYPAD "8"
            NO CHECK;
            RETURN "SKIP" TO WK_CONTROL_FIELD;
          END PROGRAM KEY;

          PROGRAM KEY IS KEYPAD "9"
            CHECK;
            RETURN "DONE" TO WK_CONTROL_FIELD;
          END PROGRAM KEY;

          PROGRAM KEY IS KEYPAD "4"
            CHECK;
            RETURN "EXIT" TO WK_CONTROL_FIELD;
          END PROGRAM KEY;

    OUTPUT  EDUC_UNIVERSITY  TO UNIVERSITY,
            EDUC_START_DATE  TO START,
            EDUC_STOP_DATE   TO STOP,
            EDUC_DEGREE      TO DEGREE;
```

(continued on next page)

Figure 11-4: Using PRKs to Allow Operator Control of Application Flow

```
   INPUT   UNIVERSITY TO EDUC_UNIVERSITY,
           START      TO EDUC_START_DATE,
           STOP       TO EDUC_STOP_DATE,
           DEGREE     TO EDUC_DEGREE;
   END DEFINITION;
```

Figure 11-4: Using PRKs to Allow Operator Control of Application Flow (Cont.)

Note that you must use the KEYPAD MODE IS APPLICATION instruction for the application program to recognize data entered on the keypad as an application program key (rather than numeric data). However, RDU does not check when you create a request that you specify the KEYPAD MODE IS instruction if you use a keypad key as a PRK.

11.4.2 Using a PRK to Return a Value to a Control Value

You can use a program request key to return a value to a control value in a conditional instruction.

In Figure 11-5, for instance, the operator can press one of two PRKs at run time (keypad 8 or keypad 4) and place predetermined values (MORE or DONE) in the control value ACTION_TO_TAKE. In a subsequent call to this same request, TDMS can evaluate the control value ACTION_TO_TAKE and then execute the appropriate request instructions.

DEPT_LABOR_REQUEST

```
FORM   IS DEPTLABOR_FORM;
RECORD IS DEPTLABOR_WORKSPACE;
RECORD IS LABOR_RECORD;

CLEAR SCREEN;
DISPLAY FORM DEPTLABOR_FORM;

CONTROL FIELD IS ACTION_TO_TAKE

        NOMATCH:
           INPUT   NUMBER  TO LABOR_EMPL_NUMBER,
                   NAME    TO LABOR_NAME,
                   DEPT    TO LABOR_DEPT;

        "MORE":
           OUTPUT
                  LABOR_EMPL_NUMBER     TO NUMBER,
                  LABOR_NAME            TO NAME,
                  LABOR_DEPT            TO DEPT;
END CONTROL FIELD;

INPUT   PROJECTNO TO WK_PROJECT_NO,
        HOURS     TO WK_HOURS,
        CODE      TO WK_OPCODE;
```

Figure 11-5: Using PRKs in Conditional Instructions

```
    KEYPAD MODE IS APPLICATION;

        PROGRAM KEY IS KEYPAD "8"
            CHECK;
            RETURN "MORE" TO ACTION_TO_TAKE ;
        END PROGRAM KEY;

        PROGRAM KEY IS KEYPAD "4"
            CHECK;
            RETURN "DONE" TO ACTION_TO_TAKE;
        END PROGRAM KEY;
    END DEFINITION;
```

Figure 11-5: Using PRKs in Conditional Instructions (Cont.)

Note that, when the request DEPT_LABOR_REQUEST is first called, the control value ACTION_TO_TAKE is blank. TDMS executes the INPUT instructions in the base section and in the NOMATCH case value. It collects basic employee information (NUMBER, NAME, AND DEPT) and project information (PROJECTNO, HOURS, and CODE).

The operator can press one of two PRKs:

- The keypad 8 key, indicating there is more information to enter on the same employee.

  TDMS returns the string "MORE" to the control value and terminates the request call. The program calls the DEPT_LABOR_REQUEST a second time. TDMS executes the request instructions following the case value "MORE". It displays the number, name, and department information. TDMS also executes the INPUT TO instruction in the base section of the request and collects additional project-related data on the same employee.

- The keypad 4 key, indicating there is no more information to enter on this employee.

  TDMS returns the string "DONE" to the control value and terminates the request call. The program issues a second call to the DEPT_LABOR_REQUEST. There is no match between the control field containing "DONE" and the case values. TDMS, therefore, executes the NOMATCH instructions. It collects employee information (NUMBER, NAME, and DEPT) and project data (PROJECTNO, HOURS, and CODE) for a different employee.

Insert second divider with text: Creating Request Libraries

Part Two:

Working with Request Libraries 12

After you successfully create a request or a series of requests in the CDD, you can create a request library definition. A request library definition allows you to identify, in one place, all the requests that are used in an application or a portion of an application. From the request library definition, you then build a request library file that the application program accesses when it wants to call a request.

This chapter explains how you use the Request Definition Utility (RDU) to:

- Create and modify request library definitions
- Build request libraries

12.1 Creating a Request Library Definition

You can create the request library definition, like the request, within RDU or in a separate text or command file. As with requests, forms, and records, you can assign a given, a full, or a relative CDD path name to the request library definition. RDU translates the given name into the full CDD path name and stores the request library definition in that CDD directory:

```
RDU> CREATE LIBRARY EMPLOYEE_LIBRARY
```

The request library definition you create is stored in your default CDD directory. The two main instructions in a request library definition are:

- The REQUEST IS instruction, which identifies all the requests you want to list in this request library.
- The FILE IS instruction, which names the VMS request library file (RLB) that you will subsequently create. (You can specify the RLB name in the BUILD LIBRARY command rather than using the FILE IS instruction in the request library definition. See the section entitled Building a Request Library File for more information.)

The following is an example of a request library definition:

```
RDU> CREATE LIBRARY EMPLOYEE_LIBRARY
RDUDFN>   REQUEST IS EMPLOYEE_MENU_REQUEST;
RDUDFN>   REQUEST IS EMPLOYEE_ADD_REQUEST;
RDUDFN>   REQUEST IS EMPLOYEE_DISPLAY_REQUEST;
RDUDFN>   FILE IS "EMPLOYLIB.RLB";
RDUDFN>   END DEFINITION;
```

Note that the RLB file name must be enclosed in quotation marks and conform to the rules for correct VMS file names. RDU stores this request library definition in the CDD unless:

- One of the requests you name is not in the CDD (only in VALIDATE mode)
- You make syntax errors

If RDU is in validate mode, it tells you which request it cannot find. You should check to see that you have entered the correct names for your requests. A spelling or typing error can result in RDU being unable to find a request in the CDD. If you have made an error, you can use the EDIT command (as described in Chapter 2) to edit your last command.

You can change the text of any of the requests named in the CDD request library definition without changing the request library definition. Because the library definition contains only the names of your requests, it is not affected by changes you make to text in those individual requests.

12.2 Copying a Request Library Definition

You can use the COPY LIBRARY command if you wish to move your request library definition to another directory in the CDD. Note that if you change the CDD directory of the request library definition, you must do one of the following:

- Move the requests named in the request library definition to the same directory
- Use a full CDD path name to identify those requests within the request library definition, if they remain in a different CDD directory

For more information on the COPY LIBRARY command, see the *VAX TDMS Reference Manual*.

12.3 Listing a Request Library Definition

You can list a request library definition. The LIST LIBRARY command allows you to see the date and time stamp of the request library definition, the names of the requests contained in it, and the RLB file specification (if any).

Use the LIST LIBRARY command to look at the contents of the request library definition:

```
RDU> LIST LIBRARY EMPLOYEE_LIBRARY
```

The LIST LIBRARY command is particularly useful when you create, modify, or replace a request library definition in Novalidate mode. You can list the contents of the request library definition and check for errors before you try to validate your request library definition or build your request library file.

12.4 Modifying a Request Library Definition

After you have created a request library definition, you can use the MODIFY LIBRARY command to change its contents. For example:

```
RDU> MODIFY LIBRARY EMPLOYEE_LIBRARY
```

This is particularly useful if you:

- Misspell one of the request names
- Need to change the given name of a request to a full CDD path name because the request is not in the same CDD directory
- Omit a request that you need
- Include a request that you do not need
- Decide to change, omit, or include a FILE IS instruction

Note that the MODIFY LIBRARY command invokes your VMS editor just as the MODIFY REQUEST command does. After making your changes, enter the appropriate command to exit your VMS editor.

12.5 Validating a Request Library Definition

You can validate the request library definition before you build the corresponding request library file if you:

- Create, modify, or replace one or more of the requests contained in the request library definition in Novalidate mode
- Change a form referred to by one or more of those requests
- Change a record referred to by one or more of those requests

You can do this at the RDU> prompt by issuing a VALIDATE LIBRARY command:

```
RDU> VALIDATE LIBRARY EMPLOYEE_LIBRARY
```

See Chapter 2, Using the Request Definition Utility (RDU), for more information on Validate mode.

12.6 Deleting a Request Library Definition

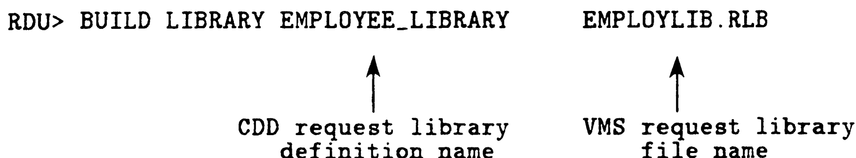
You can delete a request library definition from the CDD by using the DELETE LIBRARY command:

```
RDU> DELETE LIBRARY EMPLOYEE_LIBRARY
```

The DELETE LIBRARY command removes the request library definition from the CDD. It does not affect either the request library file or the requests the request library definition contains.

12.7 Building a Request Library File

The CDD provides an efficient and convenient way to store the definitions you create for a TDMS application. You can access them easily and make corrections and changes as needed. A TDMS application program, however, cannot access the CDD directly. A program can access the request instructions much more quickly from a file either in your directory or another VMS directory. You must create this file containing all the requests and their related form and record information. You do this by using the BUILD LIBRARY command to build a request library file. For example:



RDU searches the CDD for the request library definition, EMPLOYEE_LIBRARY, that you specify in the BUILD LIBRARY command. It extracts the requests listed in that library from the CDD and places them in the request library file, EMPLOYLIB.RLB. Note that the name RDU assigns to the new request library file is either:

- The name you specify in the FILE IS instruction, in the request library definition
- The name you specify in the BUILD LIBRARY command

You can omit the FILE IS instruction in a request library definition if you specify a file when you build the request library file. You do not have to specify the file both within the request library definition and in the BUILD LIBRARY command.

Usually you name the request library file (RLB) name in only one place. If you specify the name in both places, the BUILD LIBRARY command overrides the FILE IS instruction.

After you build a request library file, the application program can access the requests and the form and record references directly from that file.

RDU can complete a successful build only if:

- All requests listed in the CDD request library definition exist in the CDD
- All forms and records referred to in the listed requests exist in the CDD
- All mapping instructions within the requests contain correct references and mapping instructions
- You specify the name of the request library file either in the request library definition or in the BUILD LIBRARY command

RDU protects you, therefore, against placing requests that contain incorrect mapping instructions or that refer to forms or records that do not exist in the CDD in a request library file. If the BUILD command fails, RDU:

1. Displays an error message to tell you which request is incorrect or that a request library file name was not specified
2. Returns you either to the RDU> prompt or to the DCL prompt, if you entered the BUILD LIBRARY command at DCL level
3. Does not attempt to build a request library file

Part Three:

Insert third divider with text: Writing Application Programs

Introduction to TDMS Programming 13

TDMS is designed to make it easier to program an interactive forms application. TDMS does this by providing you with utilities that let you define all the terminal I/O and screen forms outside the program, in forms and requests. Consequently, you can focus your attention on designing the application and writing clear, modular program code.

Before you try to write a TDMS application program, you should know about the following TDMS elements:

- Requests and request library files
- Form definitions
- Record definitions
- Programming calls

Because TDMS stores form definitions, requests, and request library definitions in the VAX Common Data Dictionary (CDD), you should also understand the organization of the CDD. For information on the CDD, refer to the *VAX Common Data Dictionary Utilities Reference Manual*.

13.1 TDMS Programming Calls

To use requests in an application program, you must include TDMS calls in the program code. You use five of the programming calls most often in a TDMS application program; the rest of the calls provide further capabilities for a program. TDMS programming calls provide the following routines and services:

- Opening and closing request library files to access requests
- Opening and closing channels to the terminal for input and output

- Executing requests to:
  - Display forms.
  - Transfer data between a form and/or a request and a program record. That is, TDMS converts form field data to data types of receiving record fields on input and, on output, converts data types of record fields to form field data.
- Signaling TDMS return status and any extended status
- Writing or reading text from the reserved message line on the terminal
- Tracing the action of a request at run time
- Canceling a call in progress
- Copying a form to:
  - A file you specify
  - The file defined by the logical TSS$HARDCOPY

TDMS provides both synchronous and asynchronous programming calls. When you use a synchronous call, the call will complete before control returns to the application program.

The asynchronous calls are intended for the sophisticated application designed by an experienced programmer. When you use an asynchronous call, the call is initiated and control returns immediately to the application program.

However, the actions that the synchronous and asynchronous calls perform are essentially the same.

13.2 General Format of TDMS Programming Calls

TDMS programming calls conform to the VAX Procedure Calling and Condition Handling Standard. For synchronous calls, TDMS returns a standard VAX condition code to the program after a call completes, so you should include code in the program to test the return status.

For asynchronous programming calls, TDMS returns a standard VAX condition code which indicates only that the call was initiated, not that it was completed. You can test this return status by including code in your application program. Asynchronous programming calls also return a special return status block (rsb) parameter. You will learn more about this status block in Chapter 19, Using Asynchronous Calls.

Using the Primary TDMS Synchronous Calls 14

This chapter describes the general syntax and uses of the five primary TDMS synchronous calls and an additional call that lets you process the TDMS return status. The main functions of the primary calls are:

- Opening a request library file
- Opening a channel to the terminal for input and output
- Executing a request to display a form and transferring data between a form and a program record
- Closing the request library file
- Closing the channel to the terminal

There are two general concepts about TDMS synchronous calls that you should understand:

- Each call will complete before control returns to the program.
- TDMS returns a standard VAX condition code to the program after a call, so you should define a variable to receive the return status code.

Examples of the calls are presented in VAX BASIC, VAX COBOL, and VAX FORTRAN. However, you can understand the general format for most languages by studying the VAX BASIC syntax. The calls are presented in the order that you would use them in a program.

Note

In TDMS call examples, all parameters and all passing mechanisms are shown. In the calls where there are optional parameters, it is noted in the discussion.

Following the presentation of the calls, there is a discussion of how to test for the return status code. A simple program at the end of the chapter illustrates all of the primary calls.

14.1 Opening a Request Library File - TSS$OPEN_RLB

A request library file is a file containing one or more requests and the form and record information necessary to execute the requests. At run time, TDMS uses this file to access a request.

TSS$OPEN_RLB should be the first call in a program, because if the request library file does not exist or is not accessible, TDMS cannot use requests to transfer data between the form and program. The code to open a request library file is as follows.

BASIC

```
Return_status = TSS$OPEN_RLB (Request_library_file BY DESC, &
                              Library_id BY REF)
```

COBOL

```
CALL "TSS$OPEN_RLB"
     USING BY DESCRIPTOR Request-library-file,
           BY REFERENCE Library-id,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$OPEN_RLB (%DESCR(Request_library_file),
1                              %REF(Library_id))
```

Request-library-file is a variable name that contains the name of the request library file to open. You can also pass the name of the request library file in quotation marks. The default file type is .RLB. This is a required parameter.

Library-id is a variable to receive the unique number that identifies the request library file. Be sure to use different names for the library-id if you open more than one request library in a single program. This is a required parameter.

14.2 Opening a Channel - TSS$OPEN

You must open a channel to the terminal before you can perform a TDMS input or output call on that terminal. TDMS assigns a unique number for each channel that you open. The program must then pass that channel number to future TDMS calls to identify which terminal to use. The number of terminals allowed to be simultaneously open during the execution of a TDMS application is 1024.

Note

The channel number returned is not the VMS channel number returned by the SYS$ASSIGN service. Input/output calls to other services (such as VAX RMS or the $QIO system routines) *should not* be issued to the terminal.

The code to open a channel for input and output is as follows.

BASIC

```
Return_status = TSS$OPEN (Channel BY REF, &
                          Device BY DESC)
```

COBOL

```
CALL "TSS$OPEN"
     USING BY REFERENCE Channel,
           BY DESCRIPTOR Device,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$OPEN (%REF(Channel),
1                          %DESCR(Device))
```

Channel is the channel number that TDMS assigns to the terminal. You must define a variable to receive the channel number. It is a required parameter.

Device is a variable name that contains the name of the device you want to open. It is an optional parameter; SYS$INPUT is the default device. You can also pass the name of the device in quotation marks.

14.3 Transferring Data and Displaying Forms - TSS$REQUEST

You can transfer data between a program record and a form by using a request created in RDU. Requests handle all of the input and output and eliminate the need for you to code the interaction with the form in a program. When the program issues a request call, TSS$REQUEST reads and executes the instructions in the request. The code to issue a request call is as follows.

BASIC

```
Return_status = TSS$REQUEST (Channel BY REF,        &
                             Library_id BY REF,     &
                             Request_name BY DESC,  &
                             Record_1 BY REF,       &
                             Record_2 BY REF,       &
                             Record_n BY REF)
```

COBOL

```
CALL "TSS$REQUEST"
     USING BY REFERENCE Channel,
           BY REFERENCE Library-id,
           BY DESCRIPTOR Request-name,
           BY REFERENCE Record-1,
           BY REFERENCE Record-2,
           BY REFERENCE Record-n,
     GIVING Return-status.
```

FORTRAN

```
 Return_status= TSS$REQUEST (%REF(Channel),
1                           %REF(Library_id),
2                           %DESCR(Request_name),
3                           %REF(Record_1),
4                           %REF(Record_2),
5                           %REF(Record_n))
```

Channel is the channel number assigned by TDMS on the TSS$OPEN call. This is a required parameter.

Library-id is the unique number assigned to the request library file on the TSS$OPEN_RLB call, and identifies the request library file to use for this request. This is a required parameter.

Request-name is a variable that contains the name of the request. you can also pass the name of the request in quotation marks. This is a required parameter.

The record parameter corresponds to the record name or names in the RECORD IS instruction in the request. It is an optional parameter, and there can be more than one record. However, you can omit the record parameter *only if* there is no RECORD IS instruction in the request.

You include the record parameter when a request contains a RECORD IS instruction. The structure of the records must be the same as the structure of the records referred to in the request, and the order of the records passed must match the order of the records in any RECORD IS instructions in the request. In addition, the order, data type, and length of fields in the records named in the request and the records in the application program must be compatible according to the TDMS mapping rules. (See the VAX TDMS Input and Output Mapping Tables in the *VAX TDMS Reference Manual* for a complete discussion of TDMS mapping rules).

There are three important rules you must remember when passing records to the request:

1. Record definitions in an application program must be compatible with record definitions referenced in a request.
2. CDDL record definitions must be compatible with your programming language.
3. Because you pass records by reference, TDMS has no way of validating that you are passing the correct records.

If your programming language supports the extraction of records from the CDD, record structure should not be a problem. This is because you can compile your application program using the same version of the record definition that TDMS used to build the request library file.

Note that if you use CDDL or DATATRIEVE to define your records, you must pass a variable name to TSS$REQUEST that corresponds to the top level structure name in the record definition.

With languages that do not support the CDD, you have to be careful to define your records so that they are compatible with your programming language or you may generate unexpected results at run time. Chapter 16, Using Record Definitions, contains a discussion of how to define records if your programming language does not support the CDD.

14.4 Closing the Request Libary File - TSS$CLOSE_RLB

When you finish using the requests in a request library file, close the request library file with the TSS$CLOSE_RLB call. The code to close a request library file is as follows.

BASIC

```
Return_status = TSS$CLOSE_RLB (Library_id BY REF)
```

COBOL

```
CALL "TSS$CLOSE_RLB"
     USING BY REFERENCE Library-id,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$CLOSE_RLB (%REF(Library_id))
```

Library-id is the unique number assigned to the request library file on the TSS$OPEN_RLB call. It identifies which request library file to close. It is a required parameter.

14.5 Closing a Channel - TSS$CLOSE

When you finish using a channel opened by TDMS, close the channel with the TSS$CLOSE call. After you close a channel, you cannot issue any more TDMS input or output calls on that channel. If you want to use TDMS on that terminal again, you must issue another TSS$OPEN call on that terminal. TSS$CLOSE releases all TDMS resources associated with that terminal. The code to close a channel is as follows.

BASIC

```
Return_status = TSS$CLOSE (Channel BY REF, &
                           Clear_screen BY REF)
```

COBOL

```
CALL "TSS$CLOSE"
     USING BY REFERENCE Channel,
           BY REFERENCE Clear-screen,
     GIVING Return-status.
```

FORTRAN

```
Return_status = TSS$CLOSE (%REF(Channel),
1                          %REF(Clear_screen))
```

Channel is the unique number assigned to the terminal on the TSS$OPEN call. It identifies which channel to close. It is a required parameter.

Clear-screen is a variable name that, when equal to 1, clears the screen. It is an optional parameter; an uncleared screen is the default when the parameter is omitted or included and equal to 0.

14.6 Testing the Return Status Code

Each TDMS synchronous call returns a standard VAX condition code, as described in the *Introduction to VAX/VMS System Routines Manual*. When you use a TDMS call in an application program, you should check the return status code to be sure that the call completes successfully before you issue any more TDMS calls on the channel. If a TDMS call does not complete successfully and you have not made provisions to check the error, the results are unpredictable.

To check a return status, you must examine the first bit in the 32-bit status. If the first bit is set, the call was successful and the return status represents a success or informational level status code. If the first bit is not set, the call was not successful, and the return status represents a warning, error or fatal level status code.

When you test the return status and find that a call is not successful, you can issue the TSS$SIGNAL call to signal the return status and any extended status information to the terminal. TSS$SIGNAL has no parameters, so when you issue this call, it refers to the last TDMS call in the program on any channel. If you issue TSS$SIGNAL as the first call in a program, the return status code of TSS$_NORMAL is returned.

TSS$SIGNAL is useful for TDMS calls that have an extended status (that is, calls using other facilities) because in addition to the TDMS status, you get the status from the other facilities (such as VAX RMS). If a TDMS call does not have extended status information, you see the single status as if you used the VAX Run-Time Library routine LIB$SIGNAL.

The following example shows an extended status that may be returned if a TSS$OPEN_RLB call fails because the request library file referenced in the call does not exist. In this instance, when you use TSS$SIGNAL to display the extended return status, you see the following:

```
%TSS-F-ERROPNRLB, error opening request library
-RMS-E-FNF, file not found
-SYSTEM-W-NOSUCHFILE, no such file
```

To test the return status in a BASIC program, you can:

1. Declare a LONGWORD integer variable to receive the return status code
2. Declare all calls as external integer functions
3. Write a routine to test the return status

The following BASIC program segment shows how you can test the return status in an application program. In this example, if the TDMS call does not complete successfully, processing of the program ends with the call to TSS$SIGNAL. You can use the same routine to test the return status for each TDMS call.

BASIC

```
400     DECLARE INTEGER                         &
                Return_status,                  &
                Channel

500     EXTERNAL INTEGER FUNCTION               &
                TSS$OPEN,                       &
                TSS$OPEN_RLB,                   &
                TSS$REQUEST,                    &
                TSS$CLOSE_RLB,                  &
                TSS$CLOSE,                      &
                TSS$SIGNAL
```

(continued on next page)

```
          .
          .
          .

1010    Return_status = TSS$OPEN (Channel BY REF)

        GOSUB Check_return_status
          .
          .
          .

19500   Check_return_status:

        IF      (Return_status AND 1%) <> 0%
        THEN    RETURN
        ELSE    Return_status = TSS$SIGNAL
        END IF
          .
          .
          .
```

To test the return status in a COBOL program:

1. Define a variable in WORKING-STORAGE to receive the return status code
2. Write a subroutine to test the return status

The following COBOL program segment shows how you can test the return status. In this example, if the call to TDMS does not complete successfully, the program ends with a call to TSS$SIGNAL.

COBOL

```
WORKING-STORAGE SECTION.
          .
          .
          .
01  Return-status           PIC S 9(5) COMP.
01  Channel                 PIC S 9(5) COMP.
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
        CALL "TSS$OPEN" USING
             BY REFERENCE Channel
             GIVING Return-status.

        PERFORM 900-CHECK-RETURN-STATUS.
          .
          .
          .
```

```
900-CHECK-RETURN-STATUS.

        IF Return-status IS FAILURE
           CALL "TSS$SIGNAL" GIVING Return-status.
           .
           .
           .
```

14.7 Compiling and Linking a TDMS Program

You have seen all of the primary calls and learned how to test the return status that TDMS returns when you issue a TDMS call. To run any TDMS program, you have to compile it and then link it with TDMS. The following sections describe how to compile and link a TDMS program.

14.7.1 Compiling a TDMS Program

You compile a TDMS program just as you do any other program. For VAX BASIC, you issue the following command at DCL level:

```
$ BASIC program-name
```

For VAX COBOL, you issue the following command at DCL level:

```
$ COBOL program-name
```

For VAX FORTRAN, you issue the following command at DCL level:

```
$ FORTRAN program-name
```

When you compile a program, the compiler generates another file in your default directory with the same name and the file type .OBJ.

14.7.2 Linking a TDMS Program

The VAX TDMS program interface is a shareable image named TSSSHR.EXE. When the VAX TDMS software is installed, TSSSHR.EXE is placed in the shareable image symbol table library, SYS$LIBRARY:IMAGELIB.OLB.

When you link any program, the VAX Linker by default searches SYS$LIBRARY:IMAGELIB.OLB. Consequently, you can link a TDMS program simply by issuing the following command at DCL level:

```
$ LINK program-name
```

When you issue this command:

1. The linker searches IMAGELIB.OLB
2. All references to TDMS calls are resolved
3. The linker generates another file in your default directory with the same name and the file type .EXE

14.8 Two Simple TDMS Programs

The following are two simple TDMS application programs, one written in BASIC and the other in COBOL, that use all of the primary TDMS calls. This program uses the request in Figure 14-1 to display a form. Note that the program uses no records.

```
EMPLOYEE_INITIAL_REQUEST

DESCRIPTION /* Displays descriptive text.
              Explains how to use sample.
                - Collects no data -    */;

 FORM IS EMPLOYEE_INITIAL_FORM;

 CLEAR SCREEN;
 DISPLAY FORM    EMPLOYEE_INITIAL_FORM;

 WAIT;
   !The WAIT instruction allows the operator
   !to view the screen until the RETURN key
   !is pressed by the operator.

END DEFINITION;
```

Figure 14-1: Request That Displays a Form

BASIC

```
150     DECLARE INTEGER                               &
                Return_status,                        &
                Channel,                              &
                Library_id

        DECLARE INTEGER CONSTANT Clear_screen = 1%

200     EXTERNAL INTEGER FUNCTION                     &
                TSS$OPEN_RLB,                         &
                TSS$OPEN,                             &
                TSS$REQUEST,                          &
                TSS$CLOSE,                            &
                TSS$CLOSE_RLB                         &
                TSS$SIGNAL
```

```
250     !+
        ! Open the request library file; it is a good practice to open
        ! the request library file first.
        !-

        Return_status = TSS$OPEN_RLB               &
                        ("Employee.rlb",           &
                         Library_id BY REF)

        GOSUB Check_return_status


300     !+
        ! Open the channel to the terminal for input and output;
        ! you must issue this call before you can issue a TDMS input or
        ! output call on the channel.
        !-

        Return_status = TSS$OPEN (Channel BY REF)

        GOSUB Check_return_status

500     !+
        ! Issue the request call; the initial request displays a form.
        ! There are no records used by the request, so none are included
        ! on the call.
        !-

        Return_status = TSS$REQUEST                    &
                        (Channel BY REF,               &
                         Library_id BY REF,            &
                         "EMPLOYEE_INITIAL_REQUEST")

        GOSUB Check_return_status


600     !+
        ! Close the TDMS channel; note that in order to issue any more
        ! TDMS input or output calls on this channel, you have to
        ! reopen it.
        !-

        Return_status = TSS$CLOSE                      &
                        (Channel BY REF,               &
                         Clear_screen BY REF)

        GOSUB Check_return_status

700     !+
        ! Close the request library file; note that the only parameter is
        ! the library-id.
        !-

        Return_status = TSS$CLOSE_RLB                  &
                        (Library_id BY REF)

        GOSUB Check_return_status

800     GOTO 32767
```

(continued on next page)

```
19000   !+
        ! The subroutine to test the return status from a TDMS call is as
        ! follows.
        !-

     Check_return_status:

        IF      (Return_status AND 1%) <> 0%
        THEN    RETURN
        ELSE    Return_status = TSS$SIGNAL
        END IF

32767   END
```

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Simpleprg.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Return-status            PIC S9(5) COMP.
01 Library-id               PIC S9(5) COMP.
01 Request-library-file     PIC X(19) VALUE
                             "Employee.rlb".
01 Clear-screen             PIC S9(5) COMP VALUE 1.
01 Channel                  PIC S9(5) COMP.

PROCEDURE DIVISION.

100-OPENS.

* Open the request library file; it is a good practice to open the
* request library file first.


         CALL "TSS$OPEN_RLB" USING
              BY DESCRIPTOR Request-library-file,
              BY REFERENCE Library-id
              GIVING Return-status.

* Routine to test the return status.

         PERFORM 900-CHECK-RETURN-STATUS.

* Open the channel to the terminal for input and output; you must issue
* this call before you can issue a TDMS input or output call on the channel.

         CALL "TSS$OPEN" USING
              BY REFERENCE Channel
              GIVING Return_status.

* Routine to test the return status.

         PERFORM 900-CHECK-RETURN-STATUS.
```

```
300-REQUEST.

* Issue the request call; the initial request displays a form.
* No records are used, thus none appear on the call.

         CALL "TSS$REQUEST" USING
              BY REFERENCE Channel,
              BY REFERENCE Library-id,
              BY DESCRIPTOR Employee-initial-request
              GIVING Return_status.

* Routine to test the return status.

         PERFORM 900-CHECK-RETURN-STATUS.

500-CLOSE.

* Close the TDMS channel; in order to issue any more TDMS
* input or output calls on this channel, you must reopen it.

         CALL "TSS$CLOSE" USING
              BY REFERENCE Channel,
              BY REFERENCE Clear-screen
              GIVING Return-status.

* Routine to test the return status.

         PERFORM 900-CHECK-RETURN-STATUS.

* Close the request library file; note that the only parameter
* is the request library id.

         CALL "TSS$CLOSE_RLB" USING
              BY REFERENCE Library-id
              GIVING Return-status.

* Routine to test the return status.

         PERFORM 900-CHECK-RETURN-STATUS.

         GO TO 950-STOP.

* Routine to test the return status.


900-CHECK-RETURN-STATUS.

         IF Return-status IS FAILURE
            CALL "TSS$SIGNAL" GIVING Return-status.

950-STOP.

        STOP RUN.
```

Using Supplementary Calls 15

In addition to the five primary calls, TDMS provides a set of supplementary calls you can use in an application program. The supplementary calls let you:

- Read input from the terminal without using a request (TSS$READ_MSG_LINE)
- Send messages to the terminal without using a request (TSS$WRITE_MSG_LINE and TSS$WRITE_BRKTHRU)
- Copy the current form to another device (TSS$COPY_SCREEN)
- Cancel input/output (TSS$CANCEL)
- Assign and deassign application function keys (TSS$DECL_AFK and TSS$UNDECL_AFK)
- Trace TDMS call execution (TSS$TRACE_ON and TSS$TRACE_OFF)

Chapter 18 discusses application function keys and Chapter 17 discusses the Trace facility and the trace calls. This chapter introduces you to the rest of the supplementary calls. Examples are shown in VAX BASIC, VAX COBOL, and VAX FORTRAN.

15.1 Reading Messages from the Terminal - TSS$READ_MSG_LINE

You can use TSS$READ_MSG_LINE to place a message on the reserved message line on the terminal and read a line of data from the operator without using a request. The reserved message line is generally the last line on the terminal. This call gives the application program flexibility to notify the operator when the program detects an unusual condition and to prompt the operator for an answer. When the program issues this call, TDMS waits for a response from the operator.

The code to read a message from the message line is the following.

BASIC

```
Return_status = TSS$READ_MSG_LINE(Channel BY REF,        &
                                  Response_text BY DESC,  &
                                  Message_prompt BY DESC, &
                                  Response_length BY REF)
```

COBOL

```
CALL "TSS$READ_MSG_LINE"
     USING BY REFERENCE  Channel,
           BY DESCRIPTOR Response-text,
           BY DESCRIPTOR Message-prompt,
           BY REFERENCE Response-length,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$READ_MSG_LINE(%REF(Channel),
1                                  %DESCR(Response-text),
2                                  %DESCR(Message_prompt),
3                                  %REF(Response_length))
```

Channel is the channel number assigned by TDMS on the TSS$OPEN call. It is a required parameter.

Response-text is a variable that identifies a string to receive the response. It is a required parameter.

Message-prompt is a variable that contains the message you can put on the message line to prompt the operator for a reply; it is an optional parameter. You can also pass the message prompt in quotation marks.

Response-length is the address of a word to receive the length of the message line read into the buffer; it is an optional parameter. Note that the combined length of the response text and the message prompt cannot exceed the width of the terminal screen.

15.2 Sending Messages to the Terminal

TDMS provides two ways for the program to send a message to the operator:

- TSS$WRITE_MSG_LINE
- TSS$WRITE_BRKTHRU

15.2.1 Writing to the Message Line - TSS$WRITE_MSG_LINE

You can issue a TSS$WRITE_MSG_LINE call to write a message to the reserved message line on the terminal without using a request. You cannot read the message line with this call. It is useful to signal the operator to input data in form fields or to send error messages to the operator.

The code to write a message on the message line using this call is the following.

BASIC

```
Return_status = TSS$WRITE_MSG_LINE (Channel BY REF,      &
                                    Message_text BY DESC)
```

COBOL

```
CALL "TSS$WRITE_MSG_LINE"
     USING BY REFERENCE Channel,
           BY DESCRIPTOR Message-text,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$WRITE_MSG_LINE (%REF(Channel),
1                                    %DESCR(Message_text))
```

Channel is the channel number that was assigned on the TSS$OPEN call. This parameter is required.

Message-text is a variable that contains the message to be displayed on the reserved message line. This parameter is required. You can also pass the message text in quotation marks.

15.2.2 Interrupting a Request or an Existing Message Line Operation - TSS$WRITE_BRKTHRU

You can also write to the reserved message line on the terminal by issuing a TSS$WRITE_BRKTHRU call. This call allows you to interrupt the current request or message line operation in order to send a message to the operator. The code to write a message on the message line using this call is the following.

BASIC

```
Return_status = TSS$WRITE_BRKTHRU (Channel BY REF,       &
                                   Message_text BY DESC, &
                                   Bell_flag BY REF)
```

COBOL

```
CALL "TSS$WRITE_BRKTHRU"
     USING BY REFERENCE Channel,
           BY DESCRIPTOR Message-text,
           BY REFERENCE Bell-flag,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$WRITE_BRKTHRU (%REF(Channel),
1                                    %DESCR(Message_text)
2                                    %REF(Bell_flag))
```

Channel is the channel number that was assigned on the TSS$OPEN call. This parameter is required.

Message-text is a variable that contains the message to be displayed on the reserved message line. This parameter is required. You can also pass the message text in quotation marks.

Bell-flag is a flag for the terminal bell; this parameter is optional. If it is set to 1, the flag will cause the terminal bell to ring when the message text is displayed. If you do not pass this parameter, or if this parameter has a value of 0, TDMS does not ring the bell.

15.3 Copying the Current Form to a Specific Device - TSS$COPY_SCREEN

You can issue a TSS$COPY_SCREEN call to copy the contents of the currently active form to the VMS file specified in the call or the file defined by the logical name TSS$HARDCOPY.

If you use TSS$HARDCOPY, it must be defined as one or more devices or file specifications. You can define the logical name:

- As a system logical name
- As a process logical name (for example, in the operator's login command file)
- As an application program definition

To use the logical TSS$HARDCOPY without issuing a TSS$COPY_SCREEN call, the operator presses the key defined with the HARDCOPY function (the PF4 key) each time he wants a copy of the contents of the currently active form. The HARDCOPY key function and TSS$COPY_SCREEN copy:

- The background text on the form

- Double-size characters (simulated)
- Double-wide characters (simulated)
- Lines (simulated)
- Boxes (simulated)

Other video attributes are ignored.

The HARDCOPY key function also copies the data that is displayed in any field at the moment when the operator presses the HARDCOPY key.

TSS$COPY_SCREEN copies any field data that has been entered during the preceding call to the request. For example, if the preceding request included a DISPLAY FORM or USE FORM instruction, the data in the form fields at the completion of that request is copied when the TSS$COPY_SCREEN call is issued.

You cannot issue a TSS$COPY_SCREEN call while input or output is active on the specified channel, as with an outstanding call to TSS$REQUEST, TSS$REQUEST_A, TSS$WRITE_MSG_LINE, or TSS$WRITE_MSG_LINE_A.

You can use the HARDCOPY key function to copy form contents *during* a request call; you can use the TSS$COPY_SCREEN call to copy form contents *between* request calls.

The code to copy the screen is the following.

BASIC

```
Return_status = TSS$COPY_SCREEN (Channel BY REF,    &
                                 File_spec BY DESC, &
                                 Append_flag BY REF)
```

COBOL

```
CALL "TSS$COPY_SCREEN"
     USING BY REFERENCE Channel,
           BY DESCRIPTOR File-spec,
           BY REFERENCE Append-flag,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$COPY_SCREEN (%REF(Channel),
1                                 %DESCR(File_spec),
2                                 %REF(Append_flag))
```

Channel is the channel number that was assigned on the TSS$OPEN call. This parameter is required.

File-spec is the VMS file specification to which the contents of the currently active form will be directed. This parameter is optional. You can also pass the file-spec in quotation marks. If you do not specify this parameter, the value of the logical TSS$HARDCOPY will be used to determine where the contents of the currently active form are directed. If you do not specify this parameter and the logical TSS$HARDCOPY is not defined, this call will do nothing. The return status will be TSS$_NOOUTFILE, an informational status.

Append-flag is a flag that determines whether to create a new version of the file or to append the copy of the screen to the latest version of the file. This parameter is optional. A new version of the output file will be created if any of the following conditions exist:

- The parameter is not present
- The flag has a value of 0
- There is no existing version of the file

Otherwise, the current contents of the screen will be appended to the latest version of an existing VMS file.

15.4 Canceling Input/Output Calls in Progress - TSS$CANCEL

You can use TSS$CANCEL to cancel an input/output call on a channel while the call is in progress. You can do this when another facility, such as VAX RMS, detects an error in your application program. This is the only way in a TDMS application to cancel an input/output call in progress.

For example, to use TSS$CANCEL, you can have the system service $QIO read messages from a mailbox. If you have an asynchronous read outstanding on the mailbox, when a message is put into the mailbox, the AST routine associated with the mailbox initiates a TSS$CANCEL.

A return status of success on TSS$CANCEL tells the program that a call is in progress on the channel. At this point, the cancel is posted. However, the cancel is *complete* only when the call you are canceling is returned to the program.

TSS$CANCEL is the only synchronous TDMS call that can be used at asynchronous system trap (AST) level without causing an error. The TDMS calls that you can cancel are:

- TSS$COPY_SCREEN and TSS$COPY_SCREEN_A
- TSS$READ_MSG_LINE and TSS$READ_MSG_LINE_A
- TSS$REQUEST and TSS$REQUEST_A
- TSS$WRITE_MSG_LINE and TSS$WRITE_MSG_LINE_A

If TSS$CANCEL cancels a request call, the results depend entirely on whether the operator has entered data on the form and pressed the RETURN key. If the operator has pressed the RETURN key, the record buffer contains all of the data. If the operator is in the process of entering data on the form and has not pressed the RETURN key, the record buffer is not updated.

If TSS$CANCEL cancels a read message line call (TSS$READ_MSG_LINE), the results are similar to canceling a request call. If the operator has not pressed the RETURN key, nothing is returned to the record buffer.

If you cancel a call, you cannot issue another TDMS call on the channel until the canceled call returns to the program. If you try to issue another TDMS call before then, the call will fail and you will get a return status indicating that there is either a cancel in progress or input/output in progress on the channel. Once the canceled call returns, you can issue further TDMS calls.

The code to cancel a TDMS input/output call is the following.

BASIC

```
Return_status = TSS$CANCEL(Channel BY REF)
```

COBOL

```
CALL "TSS$CANCEL"
     USING BY REFERENCE Channel,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$CANCEL(%REF(Channel))
```

Channel is the same channel number that was assigned on the TSS$OPEN call. It is a required parameter.

Using Record Definitions 16

One of the most important elements of a TDMS application program is the definition of the records you use to pass information back and forth to the TDMS requests. To ensure that the record definitions the program uses match the record definition(s) that RDU uses to define its mappings, you should copy the CDD record definition(s) into your program. This chapter shows you the syntax for including CDD record definitions into programs written in three languages:

- VAX BASIC
- VAX COBOL
- VAX FORTRAN

Note that you do not have to use the BASIC, COBOL, or FORTRAN statements that let you refer to CDD record definitions. However, if you choose to define the records explicitly in the source program, you must make sure that the records you define and the CDD record definition the request uses are compatible.

TDMS supports a number of VAX data types. Included in the list is every data type that VAX BASIC, VAX COBOL, and VAX FORTRAN support (except UNSIGNED QUADWORD, a data type supported by COBOL). However, TDMS does not support all data types that you can define with the Common Data Dictionary Data Definition Language Utility (CDDL). When you are defining records, be sure you consult the data type conversion table at the end of this chapter. It will help you avoid run-time data type conversion errors.

In addition to explaining how to include record definitions into your application program, this chapter also explains how to use record definitions created by other products besides CDDL, including:

- VAX Rdb/VMS
- VAX DBMS

16.1 Using CDD Record Definitions in BASIC programs

VAX BASIC lets you refer to record definitions in the CDD using the %INCLUDE statement. This eliminates the need to define the record structure in the application program itself. However, before you pass this record to TSS$REQUEST, you must use the DECLARE or MAP statement to declare a record instance.

When you compile a BASIC program, you can use the /SHOW qualifier with the /LIST qualifier to bring the record definition into the listing file of the source program. You can then print or type the listing file and see the conversion of the record definition to BASIC RECORD syntax. The syntax of the %INCLUDE statement is as follows:

%INCLUDE %FROM %CDD cdd-path-name

The parameter cdd-path-name is a quoted string containing the path name of a CDD record definition. It can be a full, a relative, or a given CDD path name. When using relative or given path names, you must have the logical name CDD$DEFAULT defined.

The %INCLUDE statement defines and names a data structure that you use in a later DECLARE or MAP statement. You can use the %INCLUDE statement anywhere in a BASIC program to refer to CDD record definitions. However, it is a good programming practice to declare record structures at the beginning of the program to make sure the record is declared before you use it.

In the following sections you see four types of CDD record definitions, the translated BASIC version of the same record definitions if you compile the program with the /SHOW and /LIST qualifiers, and the BASIC syntax to:

- Copy the record definitions from the CDD
- Declare the record instance with the MAP statement

16.1.1 Referring to a CDD Record Definition in BASIC

Figure 16-1 shows a CDDL record definition with some group structures, the BASIC source to copy the record definition, the MAP statement, and the listing file that shows the BASIC translation.

CDD Record Definition (EMPLOYEE_RECORD)

```
DEFINE RECORD EMPLOYEE_RECORD.
EMPLOYEE_RECORD STRUCTURE.
      EMPLOYEE_NUMBER          DATATYPE SIGNED LONGWORD.
   EMPLOYEE_NAME STRUCTURE.
      FIRST_NAME               DATATYPE TEXT   15.
      MIDDLE_INITIAL           DATATYPE TEXT    1.
```

Figure 16-1: Referring to a CDD Record Definition in BASIC

```
      LAST_NAME               DATATYPE TEXT   20.
   END EMPLOYEE_NAME STRUCTURE.
   EMPLOYEE_ADDRESS STRUCTURE.
      STREET                  DATATYPE TEXT   20.
      CITY                    DATATYPE TEXT   15.
      STATE                   DATATYPE TEXT    2.
      ZIP                     DATATYPE TEXT    5.
   END EMPLOYEE_ADDRESS STRUCTURE.
      SEX                     DATATYPE TEXT    1.
      BIRTH_DATE              DATATYPE TEXT    7.
END EMPLOYEE_RECORD STRUCTURE.
END EMPLOYEE_RECORD.
```

BASIC Source Program Segment to Copy and Map the Record Definition

```
100     %INCLUDE %FROM %CDD 'Employee_record'

        MAP (Employee_record_buffer) Employee_record Emp_rec
```

BASIC Translation

```
      1 100 %INCLUDE %FROM %CDD 'Employee_record'
C1    1         RECORD  EMPLOYEE_RECORD               ! UNSPECIFIED
C1    1           LONG    EMPLOYEE_NUMBER             ! SIGNED LONGWORD
C1    1           GROUP   EMPLOYEE_NAME               ! UNSPECIFIED
C1    1             STRING  FIRST_NAME  = 15          ! TEXT
C1    1             STRING  MIDDLE_INITIAL  = 1       ! TEXT
C1    1             STRING  LAST_NAME  = 20           ! TEXT
C1    1           END GROUP
C1    1           GROUP   EMPLOYEE_ADDRESS            ! UNSPECIFIED
C1    1             STRING  STREET  = 20              ! TEXT
C1    1             STRING  CITY  = 15                ! TEXT
C1    1             STRING  STATE  = 2                ! TEXT
C1    1             STRING  ZIP  = 5                  ! TEXT
C1    1           END GROUP
C1    1           STRING  SEX  = 1                    ! TEXT
C1    1           STRING  BIRTH_DATE  = 7             ! TEXT
C1    1         END RECORD
```

Figure 16-1: Referring to a CDD Record Definition in BASIC (Cont.)

There are some important features you should note about the BASIC translation:

- The record name corresponds to the field name specified in the first CDDL STRUCTURE statement.
- BASIC substitutes the keyword GROUP for any further CDDL STRUCTURE statements.
- The listing shows the BASIC data type and uses the comment character to show the CDDL data type. In BASIC, neither GROUP nor RECORD can have a data type.

16.1.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax

Figure 16-2 shows a record definition with some group and variant structures, the BASIC source to copy the record definition, the MAP statement, and the listing file that shows the BASIC translation.

CDD Record Definition (EMPL_WORKSPACE)

```
DEFINE RECORD EMPL_WORKSPACE.
EMPL_WORKSPACE STRUCTURE.
 VARIANTS.
   VARIANT.
     WK_OPERATION       DATATYPE SIGNED WORD 1.
   END VARIANT.
   VARIANT.
     WK_DELETE          DATATYPE TEXT 2.
   END VARIANT.
 END VARIANTS.

 VARIANTS.
   VARIANT.
     WK_ERR_MSG_PARTS STRUCTURE.
       WK_ERROR_MSG_1   DATATYPE TEXT 40.
       WK_ERROR_MSG_2   DATATYPE TEXT 40.
     END WK_ERR_MSG_PARTS STRUCTURE.
   END VARIANT.
   VARIANT.
     WK_ERR_MSG         DATATYPE TEXT 80.
   END VARIANT.
 END VARIANTS.

END EMPL_WORKSPACE STRUCTURE.
END EMPL_WORKSPACE.
```

BASIC Source Program Segment to Copy and Map the Record Definition

```
100     %INCLUDE %FROM %CDD 'Empl_workspace'

        MAP (Workspace_buffer) Empl_workspace Emp_wk
```

BASIC Translation

```
      1 100 %INCLUDE %FROM %CDD 'Empl_workspace'
C1    1         RECORD  EMPL_WORKSPACE                ! UNSPECIFIED
C1    1           VARIANT
C1    1           CASE
C1    1             WORD    WK_OPERATION              ! SIGNED WORD
C1    1           CASE
C1    1             STRING  WK_DELETE   = 2           ! TEXT
C1    1           END VARIANT
C1    1           VARIANT
C1    1           CASE
C1    1             GROUP   WK_ERR_MSG_PARTS          ! UNSPECIFIED
```

Figure 16-2: Referring to a CDD Record Definition Containing the VARIANTS Syntax in BASIC

```
C1      1                 STRING  WK_ERROR_MSG_1  = 40 ! TEXT
C1      1                 STRING  WK_ERROR_MSG_2  = 40 ! TEXT
C1      1               END GROUP
C1      1             CASE
C1      1               STRING  WK_ERR_MSG  = 80       ! TEXT
C1      1             END VARIANT
C1      1           END RECORD
```

Figure 16-2: Referring to a CDD Record Definition Containing the VARIANTS Syntax in BASIC (Cont.)

In the BASIC translation:

- BASIC translates the CDDL keyword VARIANTS to VARIANT
- BASIC translates the CDDL keyword VARIANT to CASE
- The record name corresponds to the field name specified in the first CDDL STRUCTURE statement
- BASIC substitutes the keyword GROUP for any further CDDL STRUCTURE statements
- The BASIC RECORD statement shows the BASIC data type and uses the comment character to show the CDDL data type. Note that in BASIC, neither GROUP nor RECORD can have a data type.

16.1.3 Referring to CDD Array Record Definitions in BASIC

You can define and store array record definitions in the CDD and include them in a BASIC program. CDDL lets you define arrays using the OCCURS and the ARRAY keywords. If you nest a CDDL array, you get two one-dimensional arrays.

Figure 16-3 shows a CDDL record definition with a nested array, the BASIC source to copy the record definition, the MAP statement, and finally the listing file that shows the BASIC translation. Following the figure, you see how to refer to the second dimension of the array PROJECT.

CDD Record Definition (PROJECT_SUMMARY_RECORD)

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY_RECORD STRUCTURE.
   PROJECT STRUCTURE ARRAY 1:10.
      PROJECT_NUMBERS DATATYPE TEXT 5.
      TOTAL_EMPLOYEES DATATYPE F_FLOATING.
      WAGE_CLASS  ARRAY 1:3 DATATYPE F_FLOATING.
   END PROJECT STRUCTURE.
END PROJECT_SUMMARY_RECORD STRUCTURE.
END PROJECT_SUMMARY_RECORD.
```

(continued on next page)

Figure 16-3: Referring to a CDD Record Definition with Nested Arrays

BASIC Source Program Segment to Copy and Map the Record Definition

```
100     %INCLUDE %FROM %CDD 'Project_summary_record'
        MAP (Project_buffer) Project_summary_record Proj
        DECLARE INTEGER Subscript1,     &
                        Subscript2
```

BASIC Translation

```
      1 225      %INCLUDE %FROM %CDD 'Project_summary_record'
C1    1          RECORD  PROJECT_SUMMARY_RECORD      ! UNSPECIFIED
C1    1            GROUP   PROJECT(9)                ! UNSPECIFIED
C1    1              STRING  PROJECT_NUMBERS  = 5    ! TEXT
C1    1              SINGLE  TOTAL_EMPLOYEES         ! F_FLOATING
C1    1              SINGLE  WAGE_CLASS(2)           ! F_FLOATING
C1    1            END GROUP
C1    1          END RECORD
..................................1

%BASIC-I-CDDADJBOU, S 2, 1:   adjusted bounds to be zero based for:
                dimension 1 of PROJECT_SUMMARY_RECORD::PROJECT
%BASIC-I-CDDADJBOU, S 2, 1:   adjusted bounds to be zero based for:
                dimension 1 of PROJECT_SUMMARY_RECORD::PROJECT::WAGE_CLASS
```

Figure 16-3: Referring to a CDD Record Definition with Nested Arrays (Cont.)

Note that:

- BASIC adjusts the bounds of the arrays to be zero based.
- Where the CDD record definition defined PROJECT as occurring 10 times, BASIC copies the definition and defines that field as PROJECT(9). The (9) refers to the upper bound of the array PROJECT, because BASIC changes all arrays to begin with element 0 rather than element 1. If you look further in the listing files, you see that the top bound of WAGE_CLASS is (2).

You can refer to the wage class field of PROJECT_SUMMARY_RECORD as:

```
PROJ::PROJECT(Subscript1)::WAGE_CLASS(Subscript2)
```

The variable, Subscript(n), is declared on the DECLARE INTEGER statement and can be used to reference the elements in the nested arrays.

If the value of the first subscript is 0 and the value of the second subscript is 2, you get the third element of WAGE_CLASS in the first occurrence of PROJECT.

In Figure 16-4 you see a record definition that BASIC interprets as a two-dimensional array.

CDD Record Definition (PROJECT_SUMMARY)

```
DEFINE RECORD PROJECT_SUMMARY.
PROJECT_SUMMARY STRUCTURE.
   START_PROGRAM  DATATYPE TEXT 5.
   CONTROL_PROJECT ARRAY 1:10 1:3 DATATYPE TEXT 3.
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_SUMMARY.
```

BASIC Program Segment to Copy and Map the Record Definition

```
100     %INCLUDE %FROM %CDD 'Project_summary'
        MAP (Proj_summary_buffer) Project_summary Wk
        DECLARE INTEGER Subscript1,      &
                        Subscript2
```

BASIC Translation

```
     2 100      %INCLUDE %FROM %CDD 'Project_summary'
C1   2          RECORD  PROJECT_SUMMARY              ! UNSPECIFIED
C1   2            STRING  START_PROGRAM  = 5         ! TEXT
C1   2            STRING  CONTROL_PROJECT(9,2) = 3 ! TEXT
C1   2          END RECORD
....................................1

%BASIC-I-CDDADJBOU, S 1, 1:   adjusted bounds to be zero based for:
                dimension 2 of PROJECT_SUMMARY::CONTROL_PROJECT
%BASIC-I-CDDADJBOU, S 1, 1:   adjusted bounds to be zero based for:
                dimension 1 of PROJECT_SUMMARY::CONTROL_PROJECT
```

Figure 16-4: Referring to a Two-Dimensional CDD Array Record Definition in BASIC

You can refer to the control project field of PROJECT_SUMMARY as:

```
WK::CONTROL_PROJECT(Subscript1,Subscript2)
```

Table 16-1, at the end of this chapter, shows the data types that TDMS supports and the corresponding VAX, CDDL, COBOL, BASIC, and FORTRAN data types.

16.2 Using COBOL to Refer to CDD Record Definitions

VAX COBOL lets you refer to record definitions in the CDD using the COPY statement. You can use the COPY statement in the following sections of a COBOL program:

- FILE SECTION
- WORKING-STORAGE SECTION
- LINKAGE SECTION

The syntax of the COBOL COPY statement is:

COPY cdd-path-name FROM DICTIONARY

When you compile a COBOL program that uses the COPY statement:

- The compiler translates the record definition you name to COBOL source text.
- The translated record definition is in terminal format if the source program containing the COPY statement is in terminal format; otherwise, the record definition is translated to ANSI format.
- The translated definition logically replaces the COPY statement, beginning with the word COPY and ending with (and including) the punctuation character period (.).
- The compiler changes the source text as it copies it if there is a REPLACING phrase. The compiler replaces each successfully matched occurrence of a text-matching argument in the source text with the corresponding replacement item.

The parameter cdd-path-name represents a full, relative, or given CDD path name specifying a CDD record definition to be copied into the source program. It can be a nonnumeric literal or a COBOL word formed according to the rules for COBOL user-defined names. If cdd-path-name is not a literal, the compiler:

- Translates hyphens in the COBOL word to underline characters
- Treats the word as if it were enclosed in quotation marks

The resulting path name must conform to all rules for forming CDD path names.

In the following sections you see four types of CDD record definitions, the translated COBOL version of the same record definitions if you compile the program with the /LIST/COPY_LIST qualifiers, and the COBOL syntax to refer to the record definitions from the CDD.

16.2.1 Referring to a CDD Record Definition in COBOL

Figure 16-5 shows a CDD record definition with some group structures, the COBOL source to copy the record definition, and the listing file that shows the COBOL translation.

CDD Record Definition (EMPLOYEE_RECORD)

```
DEFINE RECORD EMPLOYEE_RECORD.
EMPLOYEE_RECORD STRUCTURE.
      EMPLOYEE_NUMBER          DATATYPE SIGNED LONGWORD.
   EMPLOYEE_NAME STRUCTURE.
      FIRST_NAME               DATATYPE TEXT   15.
      MIDDLE_INITIAL           DATATYPE TEXT    1.
      LAST_NAME                DATATYPE TEXT   20.
   END EMPLOYEE_NAME STRUCTURE.
   EMPLOYEE_ADDRESS STRUCTURE.
      STREET                   DATATYPE TEXT   20.
      CITY                     DATATYPE TEXT   15.
      STATE                    DATATYPE TEXT    2.
      ZIP                      DATATYPE TEXT    5.
   END EMPLOYEE_ADDRESS STRUCTURE.
      SEX                      DATATYPE TEXT    1.
      BIRTH_DATE               DATATYPE TEXT    7.
END EMPLOYEE_RECORD STRUCTURE.
END EMPLOYEE_RECORD.
```

COBOL Program Segment to Copy the CDD Record Definition

```
DATA DIVISION.
WORKING-STORAGE SECTION.

COPY 'Employee_record' FROM DICTIONARY.
```

COBOL Translation

```
  6        COPY 'Employee_record' FROM DICTIONARY.
  7L      *
  8L      * CDD$TOP.EMPLOYEE_RECORD
  9L      *
 10L       01  EMPLOYEE_RECORD.
 11L           02  EMPLOYEE_NUMBER     PIC S9(9) COMP.
 12L           02  EMPLOYEE_NAME.
 13L               03  FIRST_NAME      PIC X(15).
 14L               03  MIDDLE_INITIAL  PIC X.
 15L               03  LAST_NAME       PIC X(20).
 16L           02  EMPLOYEE_ADDRESS.
 17L               03  STREET          PIC X(20).
 18L               03  CITY            PIC X(15).
 19L               03  STATE           PIC X(2).
 20L               03  ZIP             PIC X(5).
 21L           02  SEX                 PIC X.
 22L           02  BIRTH_DATE          PIC X(7).
```

Figure 16-5: Referring to a CDD Record Definition in COBOL

It is important to note that COBOL deletes the CDDL keyword STRUCTURE from the record definition. The (01) top level structure name is the same as the first CDDL structure name.

16.2.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax

Figure 16-6 shows a record definition with group and variant structures, the COBOL source to copy the record definition, and the listing file that shows the COBOL translation.

CDD Record Definition (EMPL_WORKSPACE)

```
DEFINE RECORD EMPL_WORKSPACE.
EMPL_WORKSPACE STRUCTURE.
 VARIANTS.
   VARIANT.
     WK_OPERATION       DATATYPE SIGNED WORD 1.
   END VARIANT.
   VARIANT.
     WK_DELETE          DATATYPE TEXT 2.
   END VARIANT.
 END VARIANTS.
 VARIANTS.
   VARIANT.
     WK_ERR_MSG_PARTS STRUCTURE.
       WK_ERROR_MSG_1   DATATYPE TEXT 40.
       WK_ERROR_MSG_2   DATATYPE TEXT 40.
     END WK_ERR_MSG_PARTS STRUCTURE.
   END VARIANT.
   VARIANT.
     WK_ERR_MSG         DATATYPE TEXT 80.
   END VARIANT.
 END VARIANTS.
END EMPL_WORKSPACE STRUCTURE.
END EMPL_WORKSPACE.
```

COBOL Program Segment to Copy the CDD Record Definition

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY 'Empl_workspace' FROM DICTIONARY.
```

COBOL Translation

```
   23         COPY 'Empl_workspace' FROM DICTIONARY.
   24L       *
   25L       * CDD$TOP.EMPL_WORKSPACE
   26L       *
   27L        01  EMPL_WORKSPACE.
   28L            02  WK_OPERATION        PIC S9(4) COMP.
   29L            02  WK_DELETE REDEFINES WK_OPERATION PIC X(2).
   30L            02  WK_ERR_MSG_PARTS.
   31L                03  WK_ERROR_MSG_1  PIC X(40).
   32L                03  WK_ERROR_MSG_2  PIC X(40).
   33L            02  WK_ERR_MSG REDEFINES WK_ERR_MSG_PARTS PIC X(80).
```

Figure 16-6: Referring to a CDD Record Definition with the VARIANTS Syntax in COBOL

In the COBOL translation:

- COBOL translates the CDDL keyword VARIANT to the COBOL REDEFINES
- COBOL deletes the CDDL keyword STRUCTURE

16.2.3 Referring to CDD Array Record Definitions in COBOL

You can define and store array record definitions in the CDD and include them in a COBOL program. CDDL lets you define arrays using the OCCURS and the ARRAY keywords. Note that if you define a zero-based array in the CDD, and you try to copy it into a COBOL program, you will get a fatal level error message and COBOL will not create an object file. Therefore, all your arrays must be one-based. Figure 16-7 shows a record definition with a nested CDDL array, the COBOL source code to copy the record definition, and the listing file that shows the COBOL translation. Following the figure, you see how to refer to the second dimension of the array PROJECT.

CDD Record Definition (PROJECT_SUMMARY_RECORD)

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY_RECORD STRUCTURE.
   PROJECT STRUCTURE ARRAY 1:10.
      PROJECT_NUMBERS       DATATYPE TEXT       5.
      TOTAL_EMPLOYEES       DATATYPE F_FLOATING.
      WAGE_CLASS  ARRAY 1:3 DATATYPE F_FLOATING.
   END PROJECT STRUCTURE.
END PROJECT_SUMMARY_RECORD STRUCTURE.
END PROJECT_SUMMARY_RECORD.
```

COBOL Program Segment to Copy the CDD Record Definition

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY 'Project_summary_record' FROM DICTIONARY.
```

COBOL Translation

```
11          COPY 'Project_summary_record' FROM DICTIONARY.
12L        *
13L        * PROJECT_SUMMARY_RECORD
14L        *
15L         01  PROJECT_SUMMARY_RECORD.
16L             02  PROJECT                OCCURS 10 TIMES.
17L                 03  PROJECT_NUMBERS PIC X(5).
18L                 03  TOTAL_EMPLOYEES COMP-1.
19L                 03  WAGE_CLASS      COMP-1 OCCURS 3 TIMES.
```

Figure 16-7: Referring to a CDD Record Definition with Nested OCCURS Syntax in COBOL

You can refer to the wage class field of PROJECT_SUMMARY_RECORD as:

```
WAGE_CLASS(Subscript1,Subscript2)
```

Figure 16-8 shows a two-dimensional CDD array record definition, the syntax to copy the definition into the program, and the COBOL translation. After the figure, there is an explanation of the COBOL translation.

CDD Record Definition (PROJECT_SUMMARY)

```
DEFINE RECORD PROJECT_SUMMARY.
PROJECT_SUMMARY STRUCTURE.
   START_PROGRAM  DATATYPE TEXT 5.
   CONTROL_PROJECT ARRAY 1:10 1:3 DATATYPE TEXT 3.
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_SUMMARY.
```

COBOL Program Segment to Copy the CDD Record Definition

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY 'Project_summary' FROM DICTIONARY.
```

COBOL Translation

```
   20          COPY 'Project_summary' FROM DICTIONARY.
   21L        *
   22L        * PROJECT_SUMMARY
   23L        *
   24L         01  PROJECT_SUMMARY.
   25L             02  START_PROGRAM         PIC X(5).
   26L             02  CONTROL_PROJECT       PIC X(3) OCCURS 10 TIMES.
                       1
%COBOL-E-ERROR  401, (1) Invalid multidimensional OCCURS
```

Figure 16-8: Referring a Two-Dimensional CDD Array Record Definition in COBOL

Note that in PROJECT_SUMMARY, COBOL ignores the second dimension of CONTROL_PROJECT and issues an error level message. In this case, COBOL creates the object file but drops the second dimension and you get a one-dimensional array that occurs 10 times. Therefore, you cannot use a multi-dimensional array with COBOL.

Table 16-1, at the end of this chapter, shows the data types that TDMS supports and the corresponding VAX, CDDL, COBOL, BASIC, and FORTRAN data types.

16.3 Using FORTRAN to Refer to CDD Record Definitions

VAX FORTRAN lets you refer to record definitions in the CDD using the DICTIONARY statement. This eliminates the need for you to define records in an application program. However, before you pass this record to TSS$REQUEST, you should use the RECORD statement to declare a record instance.

When you compile a FORTRAN program, you can use the qualifier /SHOW=DICTIONARY with the /LIST qualifier to bring the record definition into the listing file of the source program. If the data attributes of the record definition are consistent with FORTRAN requirements, the record definition is included in the FORTRAN program. You can then print or type the listing file and see the conversion of the record definition to FORTRAN RECORD syntax. The syntax of the DICTIONARY statement is:

DICTIONARY cdd-path-name

The parameter cdd-path-name is a quoted string containing the path name of a CDD record definition. It can be a full, a relative, or a given CDD path name. When using relative or given path names, you must have the logical name CDD$DEFAULT defined.

The DICTIONARY statement can be used anywhere in a FORTRAN program that a specification statement (such as an STRUCTURE/END STRUCTURE block) is allowed. You can optionally include a /LIST qualifier with the DICTIONARY command:

```
DICTIONARY 'CDD$TOP.YOUR_DICTIONARY.YOUR_RECORD/LIST'
```

The DICTIONARY statement with the /LIST qualifier performs the same function as the FORTRAN/LIST/SHOW=DICTIONARY command. Note that the /LIST qualifier in the previous example follows cdd-path-name and occurs within the quotation marks.

When you use the DICTIONARY statement, the result is that FORTRAN names and defines a data structure that you can use in a later RECORD statement.

A CDD record definition can contain explanatory text in the CDDL DESCRIPTION IS clause. If you specify /SHOW=DICTIONARY with the FORTRAN command (or /LIST in the DICTIONARY statement), this text is included in the FORTRAN listing as comments. The programmer can use these comments to indicate the data type of each element. The punctuation for CDDL comments is the same as for other FORTRAN comments if you use the exclamation mark (!) comment character.

Even if you choose not to list the extracted record, the names, data types, and offsets of the CDD record definition are displayed in the program listing file's symbol map.

In the next sections you see four types of CDD record definitions, the translated FORTRAN version of the same record definitions if you compile the program with the /SHOW and /LIST qualifiers, and the FORTRAN syntax to:

- Copy the record definitions from the CDD
- Declare the record instance with the RECORD statement

16.3.1 Referring to a CDD Record Definition in FORTRAN

Figure 16-9 shows a record definition with some group structures, the FORTRAN source to copy the record definition, the RECORD statement, and the listing file that shows the FORTRAN translation.

CDD Record Definition (EMPLOYEE_RECORD)

```
DEFINE RECORD EMPLOYEE_RECORD.
EMPLOYEE_RECORD STRUCTURE.
      EMPLOYEE_NUMBER        DATATYPE SIGNED LONGWORD.
   EMPLOYEE_NAME STRUCTURE.
      FIRST_NAME             DATATYPE TEXT   15.
      MIDDLE_INITIAL         DATATYPE TEXT    1.
      LAST_NAME              DATATYPE TEXT   20.
   END EMPLOYEE_NAME STRUCTURE.
   EMPLOYEE_ADDRESS STRUCTURE.
      STREET                 DATATYPE TEXT   20.
      CITY                   DATATYPE TEXT   15.
      STATE                  DATATYPE TEXT    2.
      ZIP                    DATATYPE TEXT    5.
   END EMPLOYEE_ADDRESS STRUCTURE.
      SEX                    DATATYPE TEXT    1.
      BIRTH_DATE             DATATYPE TEXT    7.
END EMPLOYEE_RECORD STRUCTURE.
END EMPLOYEE_RECORD.
```

FORTRAN Program Segment to Copy the CDD Record Definition

```
      DICTIONARY 'Employee_record'
      RECORD /Employee_record/ Emp_rec
```

FORTRAN Translation

```
0001                DICTIONARY 'Employee_record'
0002   1          ! CDD Path Name "Employee_record"
0003   1          STRUCTURE /EMPLOYEE_RECORD/
0004   1              INTEGER*4 EMPLOYEE_NUMBER
0005   1              STRUCTURE EMPLOYEE_NAME
0006   1                  CHARACTER*15 FIRST_NAME
0007   1                  CHARACTER*1 MIDDLE_INITIAL
0008   1                  CHARACTER*20 LAST_NAME
0009   1              END STRUCTURE
0010   1              STRUCTURE EMPLOYEE_ADDRESS
```

Figure 16-9: Referring to a CDD Record Definition in FORTRAN

```
0011  1             CHARACTER*20 STREET
0012  1             CHARACTER*15 CITY
0013  1             CHARACTER*2 STATE
0014  1             CHARACTER*5 ZIP
0015  1         END STRUCTURE
0016  1         CHARACTER*1 SEX
0017  1         CHARACTER*7 BIRTH_DATE
0018  1     END STRUCTURE
```

Figure 16-9: Referring to a CDD Record Definition in FORTRAN (Cont.)

There are two important features you should note about the FORTRAN translation:

- The FORTRAN record structure name corresponds to the field name specified in the first CDDL STRUCTURE statement.
- The listing shows the equivalent FORTRAN data types for each field.

16.3.2 Referring to a CDD Record Definition Containing the VARIANTS Syntax

Figure 16-10 shows a record definition with some group and variant structures, the FORTRAN source to copy the record definition, the RECORD statement, and the listing file that shows the FORTRAN translation.

CDD Record Definition (EMPL_WORKSPACE)

```
DEFINE RECORD EMPL_WORKSPACE.
EMPL_WORKSPACE STRUCTURE.
 VARIANTS.
   VARIANT.
     WK_OPERATION       DATATYPE SIGNED WORD 1.
   END VARIANT.
   VARIANT.
     WK_DELETE          DATATYPE TEXT 2.
   END VARIANT.
 END VARIANTS.

 VARIANTS.
   VARIANT.
     WK_ERR_MSG_PARTS STRUCTURE.
       WK_ERROR_MSG_1   DATATYPE TEXT 40.
       WK_ERROR_MSG_2   DATATYPE TEXT 40.
     END WK_ERR_MSG_PARTS STRUCTURE.
   END VARIANT.
   VARIANT.
     WK_ERR_MSG         DATATYPE TEXT 80.
   END VARIANT.
 END VARIANTS.

END EMPL_WORKSPACE STRUCTURE.
END EMPL_WORKSPACE.
```

(continued on next page)

Figure 16-10: Referring to a CDD Record Definition Containing the VARIANTS Syntax in FORTRAN

FORTRAN Program Segment to Copy the CDD Record Definition

```
DICTIONARY 'EMPL_WORKSPACE'
RECORD /EMPL_WORKSPACE/ EMP_WK
```

FORTRAN Translation

```
0022            DICTIONARY 'Empl_workspace'
0023  1       ! CDD Path Name "Empl_workspace"
0024  1       STRUCTURE /EMPL_WORKSPACE/
0025  1           UNION
0026  1               MAP
0027  1                   INTEGER*2 WK_OPERATION
0028  1               END MAP
0029  1               MAP
0030  1                   CHARACTER*2 WK_DELETE
0031  1               END MAP
0032  1           END UNION
0033  1           UNION
0034  1               MAP
0035  1                   STRUCTURE WK_ERR_MSG_PARTS
0036  1                       CHARACTER*40 WK_ERROR_MSG_1
0037  1                       CHARACTER*40 WK_ERROR_MSG_2
0038  1                   END STRUCTURE
0039  1               END MAP
0040  1               MAP
0041  1                   CHARACTER*80 WK_ERR_MSG
0042  1               END MAP
0043  1           END UNION
0044  1       END STRUCTURE
```

Figure 16-10: Referring to a CDD Record Definition Containing the VARIANTS Syntax in FORTRAN (Cont.)

In the FORTRAN translation:

- FORTRAN translates the keyword VARIANTS to UNION
- FORTRAN translates the keyword VARIANT to MAP

16.3.3 Referring to CDD Array Record Definitions in FORTRAN

You can define and store array record definitions in the CDD and include them in a FORTRAN program. CDDL lets you define arrays using the OCCURS and the ARRAY keywords. If you nest CDDL arrays, you get a one-dimensional array within a one-dimensional array structure.

Figure 16-11 shows a record definition with a nested array, the FORTRAN source to copy the record definition, the RECORD statement, and the listing file that shows the FORTRAN translation. Following the figure, you see how to refer to the second dimension of the array PROJECT.

CDD Record Definition (PROJECT_SUMMARY_RECORD)

```
DEFINE RECORD PROJECT_SUMMARY_RECORD.
PROJECT_SUMMARY_RECORD STRUCTURE.
   PROJECT STRUCTURE ARRAY 1:10.
      PROJECT_NUMBERS DATATYPE TEXT 5.
      TOTAL_EMPLOYEES DATATYPE F_FLOATING.
      WAGE_CLASS  ARRAY 1:3 DATATYPE F_FLOATING.
   END PROJECT STRUCTURE.
END PROJECT_SUMMARY_RECORD STRUCTURE.
END PROJECT_SUMMARY_RECORD.
```

FORTRAN Program Segment to Copy the CDD Record Definition

```
      DICTIONARY 'PROJECT_SUMMARY_RECORD'
      RECORD /PROJECT_SUMMARY_RECORD/ PROJ
      INTEGER SUBSCRIPT1, SUBSCRIPT2
```

FORTRAN Translation

```
0048            DICTIONARY 'Project_summary_record'
0049   1      ! CDD Path Name "Project_summary_record"
0050   1      STRUCTURE /PROJECT_SUMMARY_RECORD/
0051   1          STRUCTURE PROJECT(1:10)
0052   1              CHARACTER*5 PROJECT_NUMBERS
0053   1              REAL*4 TOTAL_EMPLOYEES
0054   1              REAL*4 WAGE_CLASS(1:3)
0055   1          END STRUCTURE
0056   1      END STRUCTURE
```

Figure 16-11: CDDL Record Definition with Nested OCCURS Syntax

You can refer to the wage class field of PROJECT_SUMMARY_RECORD as:

```
PROJ::PROJECT(Subscript1)::WAGE_CLASS(Subscript2)
```

If the value of the first subscript is 1 and the value of the second subscript is 3, you get the third element of WAGE_CLASS.

In Figure 16-12 you see a record definition that FORTRAN interprets as a two-dimensional array.

CDD Record Definition (PROJECT_SUMMARY)

```
DEFINE RECORD PROJECT_SUMMARY.
PROJECT_SUMMARY STRUCTURE.
   START_PROGRAM                                  DATATYPE TEXT 5.
   CONTROL_PROJECT COLUMN_MAJOR ARRAY 1:10 1:3 DATATYPE TEXT 3.
END PROJECT_SUMMARY STRUCTURE.
END PROJECT_SUMMARY.
```

(continued on next page)

Figure 16-12: Referring to a Two-Dimensional CDD Array Record Definition in FORTRAN

FORTRAN Program Segment to Copy the CDD Record Definition

```
      DICTIONARY 'PROJECT_SUMMARY'
      RECORD /PROJECT_SUMMARY/ WK
      INTEGER SUBSCRIPT1, SUBSCRIPT2
```

FORTRAN Translation

```
0001                DICTIONARY 'PROJECT_SUMMARY'
0002  1            ! CDD Path Name "Employee_record"
0003  1            STRUCTURE /PROJECT_SUMMARY/
0004  1               STRUCTURE PROJECT
0005  1                   CHARACTER*5 NUMS
0006  1                   CHARACTER*3 WAGE_CLASS(1:10,1:3)
0007  1               END STRUCTURE
0008  1            END STRUCTURE
0009  1
```

Figure 16-12: Referring to a Two-Dimensional CDD Array Record Definition in FORTRAN (Cont.)

You can refer to the control project field of PROJECT_SUMMARY as:

```
WK::CONTROL_PROJECT(Subscript1,Subscript2)
```

Note that a multi-dimensional array must specify COLUMN_MAJOR or FORTRAN issues a fatal error and does not finish compiling the program.

Table 16-1, at the end of this chapter, shows the data types that TDMS supports and the corresponding VAX, CDDL, FORTRAN, BASIC, and COBOL data types.

16.4 Using Record Definitions Created by Database Management Systems

Throughout this book, the examples use record definitions created with the VAX CDD Data Definition Language (CDDL). However, there are several other products that create CDD record definitions that TDMS can use. In particular, both of the VAX Information Architecture database management systems can store records in the CDD:

- VAX Rdb/VMS
- VAX DBMS

It is important, when building applications that use these products, to make sure your application program and your TDMS requests refer to the correct record definitions. If your requests do refer to the correct record definitions, changes to the database (adding a new record field or changing the size of an existing field) can often be accomplished without changing any request or programming code. You simply rebuild the request library and recompile the application program.

The following sections explain how to refer to record definitions created by VAX Rdb/VMS and VAX DBMS.

16.4.1 Using Record Definitions Created by VAX Rdb/VMS

One of the products you can use to create record definitions in the CDD is VAX Rdb/VMS. When you define a database using Rdb/VMS, it optionally creates a hierarchy of record definitions and other CDD entities in the CDD under a path name you specify.

The records you use for mapping data from an Rdb/VMS database to a TDMS form field are the record definitions for the Rdb/VMS *relations*. These record definitions are stored in the subdirectory RDB$RELATIONS, using the name of the relation as the record name.

For example, if the Rdb/VMS database path name is RDB_DB (in your default CDD directory) and the relation you are mapping to a form is called SALARY_HISTORY, the path name for the record definition that defines that relation is RDB_DB.RDB$RELATIONS.SALARY_HISTORY. This is the path name you must use in the RECORD IS instruction in the request and in the program statement that copies the record definition. This also assumes that you have defined CDD$DEFAULT.

The following example illustrates how to use the Rdb/VMS record definition to pass information from an Rdb/VMS relation to a TDMS request. The example uses Rdb/VMS data manipulation statements for VAX BASIC.

```
                          .
                          .
                          .

          ! Declare the Rdb records.
   %INCLUDE %FROM %CDD    "Rdb_db.Rdb$relations.Salary_history"
   DECLARE Salary_history Salary_rec

          ! Invoke the database.
&RDB&    INVOKE DATABASE PATHNAME "Rdb_db"
                          .
                          .
                          .

          ! Get a record.
&RDB&    FOR S IN Salary_history
&RDB&        WITH S.Id_number = Salary_rec::Id_number
&RDB&       GET  Salary_rec::Start_date = S.Start_date
&RDB&            Salary_rec::End_date = S.End_date
&RDB&            Salary_rec::Salary = S.Salary
&RDB&       END_GET
```

(continued on next page)

```
        ! Display the record.
        Return_status = TSS$REQUEST (Tdms_channel,            &
                                     Tdms_lib_id,             &
                                     "Salary_request",        &
                                     Salary_rec BY REF)
&RDB&   END_FOR
                          .
                          .
                          .
```

In cases where you are fetching the entire record for use by TDMS, you can abbreviate the Rdb/VMS GET statement with the use of the asterisk (*) wildcard character. For example:

```
        ! Get a record.
&RDB&   FOR S IN Salary_history
&RDB&       WITH S.Id_number = Salary_rec::Id_number
&RDB&     GET  Salary_rec = S.*
&RDB&     END_GET
```

For more information on Rdb/VMS and Rdb/VMS data manipulation statements, see the *VAX Rdb/VMS Guide to Programming.*

16.4.2 Using Record Definitions Created by VAX DBMS

Another product that creates record definitions in the CDD is VAX DBMS. When you define a DBMS database, DBMS creates a hierarchy of subdirectories to contain the record definitions for the schema and its subschemas.

A DBMS database can have several subschema definitions, each providing a different logical description of the records and sets in the database. However, you can specify only one subschema when you invoke the database. That one subschema defines the structure of the records for the database transactions during that session. Consequently, you must use the records defined by that subschema when mapping record fields contained in the DBMS database to TDMS form fields.

Subschema definitions are stored in a CDD subdirectory named DBM$SUBSCHEMAS under the main DBMS schema name. The record definitions are then stored in a subdirectory named DBM$RECORDS within each subschema subdirectory. For example, if the path name for the DBMS schema is DBMS_DB (in your default CDD directory), the subschema you are using is SUBSCHEMA1, and you want to map the record PERSONNEL to a TDMS form, the path name for that record definition is:

```
DBMS_DB.DBM$SUBSCHEMAS.SUBSCHEMA1.DBM$RECORDS.PERSONNEL
```

This is the path name you must use in the RECORD IS instruction in the request and in the program statement that copies the record definition. This assumes that you have defined CDD$DEFAULT.

Note that you must define the records as work areas in your program. Although the DBMS precompiler creates its own work area for fetching and storing data from the database root file, it is safest to use a separate work area for transmitting data to and from TDMS. Consequently, as in the following example, you must move data values from the TDMS work area to the DBMS work area between calls to TDMS requests and DBMS DML statements to fetch and store data.

The following example illustrates the use of DBMS record definitions to select and update a DBMS record using TDMS forms and requests. The example uses the VAX DBMS data manipulation language (DML) for VAX BASIC:

```
          .
          .
          .

     ! Declare the TDMS work area
%INCLUDE %FROM %CDD                           &
   "DBMS_db.Dbm$subschemas.Subschema1.Dbm$records.Employee"
DECLARE Employee Employee_rec

     ! Invoke the database, specifying the subschema SUBSCHEMA1
# INVOKE Subschema1 WITHIN DBMS_db
# READY CONCURRENT UPDATE

     ! Get the name of an employee
Return_status = TSS$REQUEST (Channel,                        &
                             Library_id,                     &
                             "Get_name_request",             &
                             Employee_rec)

     ! Move the name into the DBMS work area
Emp_name = Employee_rec::Emp_name

     ! Find the appropriate record
# FETCH FIRST Employee USING Emp_name

     ! Move the DBMS data into the TDMS work area
Employee_rec::Emp_name = Emp_name
Employee_rec::Address = Address
Employee_rec::City = City
Employee_rec::State = State

     ! call the update request
Return_status = TSS$REQUEST (Channel,                       &
                             Library_id,                    &
                             "Update_request",              &
                             Employee_rec)
```

(continued on next page)

```
     ! Move the updated data back into the DBMS work area
Emp_name = Employee_rec::Emp_name
Address  = Employee_rec::Address
City     = Employee_rec::City
State    = Employee_rec::State

     ! Modify the DBMS record
# MODIFY Employee
# COMMIT
              .
              .
              .
```

For more information on VAX DBMS and the DBMS data manipulation language, see the *VAX DBMS Programming Guide* and *VAX DBMS Programming Reference Manual*.

16.4.3 Displaying and Updating Database Records in Scrolled Regions

The previous sections explain how to use VAX Rdb/VMS and VAX DBMS records for entering, displaying, and updating individual records using TDMS. Another common practice is to use TDMS scrolled regions to display and/or update a collection of records.

The nature of databases is to store each record separately. A stream of records can then be created using data manipulation statements. However, there is no single record definition that represents the entire collection.

What you must do to use the record collection in a TDMS scrolled region is:

1. Define a record that contains an array of records
2. Declare the new array record in your application program and TDMS request
3. Use data manipulation statements to create a collection of records and load the collection into the array record
4. Pass the array record to the TDMS request

The following sections discuss this process in detail, using an Rdb/VMS database as an example.

Note

Although the following sections use Rdb/VMS as an example, the process is identical for creating scrolled regions for VAX DBMS records.

16.4.3.1 Defining an Array Record -- In the Rdb/VMS example in the section entitled Using Record Definitions Created by VAX Rdb/VMS, the database (RDB_DB) contains personnel records and has a relation that defines the structure of a single record, SALARY_HISTORY, containing fields for the employee id, start and end dates, and salary. One possible application might be to display the entire salary history for one employee in a scrolled region. To do this you need a scrolled region with fields for the start date, end date, and salary for each of the employee's salary records.

The easiest way to create a record that contains an array of SALARY_HISTORY records is to use the CDDL COPY FROM statement:

TDMS_REC

```
DEFINE RECORD TDMS_REC.
  TDMS_REC STRUCTURE.
    SAL_ARRAY STRUCTURE OCCURS 50 TIMES.
      SALARY COPY FROM RDB_DB.RDB$RELATIONS.SALARY_HISTORY.
    END SAL_ARRAY STRUCTURE.
  END TDMS_REC STRUCTURE.
END TDMS_REC RECORD.
```

By using the COPY FROM statement, any changes to the database will be reflected in the TDMS requests and the application program when you rebuild the request library and recompile the program.

However, if the request uses the scrolled region for display only, you will also need a one character array field to map for input so the operator can view the scrolled region. See Chapter 10, How to Input and Display Data in a Scrolled Region, for more information on scrolled regions. This field can be added to the array record:

TDMS_REC

```
DEFINE RECORD TDMS_REC.
  TDMS_REC STRUCTURE.
    SAL_ARRAY STRUCTURE OCCURS 50 TIMES.
       SALARY COPY FROM RDB_DB.RDB$RELATIONS.SALARY_HISTORY.
          ! One character field for
          ! display-only scrolled regions.
       FIELD_FOR_DISPLAY DATATYPE IS TEXT SIZE IS 1.
    END SAL_ARRAY STRUCTURE.
  END TDMS_REC STRUCTURE.
END TDMS_REC RECORD.
```

16.4.3.2 Declaring the Array Record in the Application and TDMS Requests -- To use the new array record, you must declare it in the application program and in the TDMS request as you would any other CDD record. For example, a VAX BASIC program would require the following statements:

```
       ! Declare the TDMS array record.
   %INCLUDE %FROM %CDD  "Tdms_rec"
   DECLARE Tdms_rec Salary_rec
```

The TDMS request might look like the following:

SALARY_HIST_REQUEST

```
CREATE REQUEST SALARY_HIST_REQUEST
FORM IS SALARY_HIST_FORM;
RECORD IS TDMS_REC;

   USE FORM SALARY_HIST_FORM;

   OUTPUT ID_NUMBER  [1 TO 50] TO ID_NUMBER  [1 TO 50],
          START_DATE [1 TO 50] TO START_DATE [1 TO 50],
          END_DATE   [1 TO 50] TO END_DATE   [1 TO 50],
          SALARY     [1 TO 50] TO SALARY     [1 TO 50];

   INPUT  FIELD_FOR_DISPLAY [1 TO 50] TO
          FIELD_FOR_DISPLAY [1 TO 50];

END DEFINITION;
```

16.4.3.3 Creating a Collection of Records and Loading the Array -- Once you define the array record and the request that maps the array to a scrolled region, you can then write the program code that will load the array. What you must do is create a program loop that performs a data manipulation statement to read each record and copy it into one element of the array. This is done by using a counter that identifies which element of the array is currently being loaded and incrementing the counter by one each time the loop is executed. For the Rdb/VMS example, the program might look as follows:

```
        ! Initialize a counter for the array.
        Counter = 0

        ! Load a collection of records in the array.
&RDB&   FOR S IN Salary_history
&RDB&       WITH S.Id_number = Salary_rec::Id_number
&RDB&     GET Salary_rec::Sal_array(Loop_index)::Start_date = S.Start_date
&RDB&         Salary_rec::Sal_array(Loop_index)::End_date = S.End_date
&RDB&         Salary_rec::Sal_array(Loop_index)::Salary = S.Salary
&RDB&     END_GET
```

```
        ! Increment the counter.
        Counter = Counter + 1

&RDB&   END_FOR
```

Note that the array subscript is specified with the name of the structure Sal_array, not with the record name or the field name.

You must also be careful to initialize any array elements not loaded from the database. There are two reasons for this:

1. If the program routine that loads the array is called more than once, the remaining array elements might contain spurious data from a previous call.
2. Any uninitialized fields mapped for output can result in spurious data on the form or a TDMS data type conversion error. (Data type conversion errors commonly occur when a text field contains null characters.)

You can either initialize the entire array before loading the record collection, or you can initialize the remaining array elements after the load operation. For example, the following program fragment uses the value of the counter (set by the previous loop) to initialize the remaining array elements after the load operation:

```
        ! Initialize the remaining array elements
        FOR Loop_index = Counter to 49
            Salary_rec::Sal_array(Counter)::Start_date = SPACE$(8)
            Salary_rec::Sal_array(Counter)::End_date = SPACE$(8)
            Salary_rec::Sal_array(Counter)::Salary = 0
        NEXT Loop_index
```

16.4.3.4 Passing the Array to the Request -- Once the record collection is loaded into the array, the program can call the TDMS request passing the array record name as an argument. For example:

```
         ! Call the update request
    Return_status = TSS$REQUEST (Channel,                          &
                                 Library_id,                       &
                                 "Salary_hist_request",            &
                                 Salary_rec)
```

If the request uses the scrolled region for input, the program must then perform the reverse operation -- unloading the array record and modifying the database records associated with each array element.

16.5 Summary of Supported Data Types for Different Languages

Table 16-1 lists all VAX data types that TDMS supports. It also lists the corresponding data type names for those data types in CDDL, VAX BASIC, VAX COBOL, VAX FORTRAN.

Table 16-1: Data Type Conversion Chart

VAX Data Type	CDDL Data Type	TDMS Data Type	BASIC Data Type	COBOL Data Type	FORTRAN Data Type
SIGNED BYTE	SIGNED BYTE	SIGNED BYTE	BYTE	Unsupported	BYTE
SIGNED WORD	SIGNED WORD	SIGNED WORD	WORD	PIC S9(1-4) COMP	INTEGER*2
SIGNED LONGWORD	SIGNED LONGWORD	SIGNED LONGWORD	LONG	PIC S9(5-9) COMP	INTEGER*4 or INTEGER
SIGNED QUADWORD	SIGNED QUADWORD	SIGNED QUADWORD	Unsupported	PIC S9(10-18) COMP	Unsupported
SIGNED OCTAWORD	SIGNED OCTAWORD	Unsupported	Unsupported	Unsupported	Unsupported
UNSIGNED BYTE	UNSIGNED BYTE	UNSIGNED BYTE	Unsupported	Unsupported	Unsupported
UNSIGNED WORD	UNSIGNED WORD	UNSIGNED WORD	Unsupported	PIC 9(4) COMP	Unsupported
UNSIGNED LONGWORD	UNSIGNED LONGWORD	UNSIGNED LONGWORD	Unsupported	PIC 9(9) COMP	Unsupported
UNSIGNED QUADWORD	UNSIGNED QUADWORD	Unsupported	Unsupported	PIC 9(18) COMP	Unsupported
UNSIGNED OCTAWORD	UNSIGNED OCTAWORD	Unsupported	Unsupported	Unsupported	Unsupported
F_FLOATING	F_FLOATING	F_FLOATING	SINGLE	COMP-1	REAL or REAL*4
D_FLOATING	D_FLOATING	D_FLOATING	DOUBLE	COMP-2	REAL*8
G_FLOATING	G_FLOATING	G_FLOATING	GFLOAT	Unsupported	REAL*8
H_FLOATING	H_FLOATING	H_FLOATING	HFLOAT	Unsupported	REAL*16
F_FLOATING COMPLEX	F_FLOATING COMPLEX	Unsupported	Unsupported	Unsupported	COMPLEX or COMPLEX*8
D_FLOATING COMPLEX	D_FLOATING COMPLEX	Unsupported	Unsupported	Unsupported	COMPLEX*16
G_FLOATING COMPLEX	G_FLOATING COMPLEX	Unsupported	Unsupported	Unsupported	COMPLEX*16
H_FLOATING COMPLEX	H_FLOATING COMPLEX	Unsupported	Unsupported	Unsupported	Unsupported
CHARACTER	TEXT	TEXT	STRING	PIC X(n)	CHARACTER*n
UNSIGNED NUMERIC	UNSIGNED NUMERIC	UNSIGNED NUMERIC	Unsupported	PIC 9(m)V9(n)	Unsupported
LEFT SEPARATE NUMERIC	LEFT SEPARATE NUMERIC	LEFT SEPARATE NUMERIC	Unsupported	PIC S9(m)V9(n) LEADING SEPARATE	Unsupported
RIGHT SEPARATE NUMERIC	RIGHT SEPARATE NUMERIC	RIGHT SEPARATE NUMERIC	Unsupported	PIC S9(m)V9(n) TRAILING SEPARATE	Unsupported
LEFT OVERPUNCHED NUMERIC	LEFT OVERPUNCHED NUMERIC	LEFT OVERPUNCHED NUMERIC	Unsupported	PIC S9(m)V9(n) LEADING	Unsupported

Table 16-1: Data Type Conversion Chart (Cont.)

VAX Data Type	CDDL Data Type	TDMS Data Type	BASIC Data Type	COBOL Data Type	FORTRAN Data Type
RIGHT OVERPUNCHED NUMERIC	RIGHT OVERPUNCHED NUMERIC	RIGHT OVERPUNCHED NUMERIC	Unsupported	PIC S9(m)V9(n) TRAILING	Unsupported
ZONED NUMERIC	SIGNED NUMERIC	ZONED NUMERIC	Unsupported	Unsupported	Unsupported
PACKED DECIMAL	PACKED NUMERIC	PACKED DECIMAL	DECIMAL	PIC S9(m)V9(n) COMP-3	Unsupported
DATE	DATE	DATE	Unsupported	PIC S9(11)V9(7)	Unsupported

Notes to Table 16-1

- COBOL has no exact equivalent for the unsigned integer data types UNSIGNED WORD and UNSIGNED LONGWORD. The COBOL compiler issues a warning diagnostic and treats the item as an unsigned COMP data type.
- The FORTRAN data type INTEGER corresponds to SIGNED LONGWORD if you compile the program with the default qualifier /I4. If you compile the program with the /NOI4 qualifier, INTEGER corresponds to the TDMS data type SIGNED WORD.
- The FORTRAN data type REAL*8 corresponds to G_FLOATING if you compile the program with the /G_FLOATING qualifier; otherwise, REAL*8 corresponds to the TDMS data type D_FLOATING.
- The FORTRAN data type COMPLEX*16 corresponds to G_FLOATING COMPLEX if you compile the program with the /G_FLOATING qualifier; otherwise, COMPLEX*16 corresponds to the VAX data type D_FLOATING COMPLEX.

Debugging a TDMS Application Program 17

The Trace facility is a tool TDMS provides to help you debug a TDMS application program. Trace lets you monitor the action of a TDMS application program at run time. It is most useful as an aid to debug programs that use conditional requests because you have no way of knowing which set of instructions in a conditional request will be executed until you run an application. You can use Trace to:

- Trace the execution of a request at run time, including:
  - Transfer of data from a program record to a form
  - Transfer of data from a form to a program record
  - Values of control fields
- Trace TDMS calls, including:
  - Parameter values
  - Entry and exit time of each TDMS call

17.1 How to Enable the Trace Facility

There are two methods of enabling the Trace facility:

- Defining a logical name before you run an application program
- Issuing trace calls from an application program

In each case you can direct the trace output to a device or file specification.

17.1.1 Defining a Logical Name

You can define a logical name for the trace output before you run an application program. For example, you can issue the DCL command:

```
$  DEFINE TSS$TRACE_OUTPUT PROG1
```

When you issue this command:

- The TDMS Trace facility is turned on when the program issues the first TDMS call
- PROG1.LOG contains the trace output (note that .LOG is the default file type)
- Trace is on for as long as the program is running or until the program issues a TSS$TRACE_OFF call

When an application program stops, you can type or print PROG1.LOG to see the trace output.

17.1.2 Issuing Trace Calls from an Application Program

There are two calls you can use in an application program to enable and disable Trace:

1. TSS$TRACE_ON
2. TSS$TRACE_OFF

Each time you call TSS$TRACE_ON or TSS$TRACE_OFF, a message is written to the trace output file or device, stating the time the action occurred. In addition, TDMS attempts to translate TSS$TRACE_OUTPUT (the logical name). If TDMS cannot find a translation, the trace output defaults to DBG$OUTPUT, which in turn defaults to SYS$OUTPUT.

The code to issue the trace calls is the following.

BASIC

```
Return_status = TSS$TRACE_ON
Return_status = TSS$TRACE_OFF
```

COBOL

```
CALL "TSS$TRACE_ON" GIVING Return-status.
CALL "TSS$TRACE_OFF" GIVING Return-status.
```

FORTRAN

```
Return_status = TSS$TRACE_ON ( )
Return_status = TSS$TRACE_OFF ( )
```

There are no parameters for the trace calls. When you issue the call to TSS$TRACE_ON, TDMS turns on the Trace facility and writes a message to a trace log file, noting that Trace is on and the time the call was issued. If you want to turn Trace off during the application program, you can use the TSS$TRACE_OFF call in the program. For example, if you want to trace the action of request calls only, you can precede each request call with TSS$TRACE_ON and follow it with TSS$TRACE_OFF.

17.2 Results of Using Trace

The trace output you get is similar to Figure 17-1. Following the figure, there is an explanation of each line of the trace output.

This sample output comes from running the Employee sample application through the following procedures with Trace on. The Employee sample is available to you as an optional part of the TDMS installation procedure.

- TSS$OPEN_RLB
- TSS$OPEN
- TSS$REQUEST: the Initial request
- TSS$REQUEST: the Menu request
- TSS$REQUEST: the Add request
- TSS$REQUEST: the Menu request
- TSS$CLOSE_RLB
- TSS$CLOSE

Line numbers are inserted for ease of explanation only. You do not see line numbers when you use the Trace facility. In this example, Trace was enabled by defining a logical name.

```
1.  %TSS-I-TRARET, TSS$TRACE_ON returned on  8-NOV-1986 11:22:48.93

2.  %TSS-I-TRACALL, TSS$OPEN_RLB called on  8-NOV-1986 11:22:49.08

3.  %TSS-I-TRARET, TSS$OPEN_RLB returned on  8-NOV-1986 11:22:49.30
```

(continued on next page)

Figure 17-1: Sample Trace Output

```
 4.  %TSS-I-TRACALL, TSS$OPEN called on  8-NOV-1986 11:22:49.31

 5.  %TSS-I-TRARET, TSS$OPEN returned on  8-NOV-1986 11:22:49.38

 6.  %TSS-I-TRACALL, TSS$REQUEST called on  8-NOV-1986 11:22:49.60

 7.  %TSS-I-TRAPRMCHN, channel is 898528

 8.  %TSS-I-TRAPRMREQNAM, request name is EMPLOYEE_INITIAL_REQUEST

 9.  %TSS-I-TRARET, TSS$REQUEST returned on  8-NOV-1986 11:22:52.29

10.  %TSS-I-TRACALL, TSS$REQUEST called on  8-NOV-1986 11:22:52.29

11.  %TSS-I-TRAPRMCHN, channel is 898528

12.  %TSS-I-TRAPRMREQNAM, request name is EMPLOYEE_MENU_REQUEST

13.  %TSS-I-TRAOUTPUT,
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.FIRST_MESSAGE_FIELD of value
     "                                  " output to FIRST_MESSAGE_FIELD

14.  %TSS-I-TRAOUTPUT,
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.SECOND_MESSAGE_FIELD of value
     "                                  " output to SECOND_MESSAGE_FIELD

15.  %TSS-I-TRAINPUT, SELECTION of value "1" input to
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.SELECTION

16.  %TSS-I-TRAINPUT, EMPLOYEE_NUMBER of value "1232342" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NUMBER

17.  %TSS-I-TRARET, TSS$REQUEST returned on  8-NOV-1986 11:22:58.51

18.  %TSS-I-TRACALL, TSS$REQUEST called on  8-NOV-1986 11:22:58.85

19.  %TSS-I-TRAPRMCHN, channel is 898528

20.  %TSS-I-TRAPRMREQNAM, request name is EMPLOYEE_ADD_REQUEST

21.  %TSS-I-TRAOUTPUT,
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NUMBER of
     value "1232342" output to EMPLOYEE_NUMBER

22.  %TSS-I-TRAINPUT, FIRST_NAME of value "Jack        "
     input to EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NAME.FIRST_NAME

23.  %TSS-I-TRAINPUT, MIDDLE_INITIAL of value "R" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NAME.MIDDLE_INITIAL

24.  %TSS-I-TRAINPUT, LAST_NAME of value "Sinical             " input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NAME.LAST_NAME
```

Figure 17-1: Sample Trace Output (Cont.)

```
25.  %TSS-I-TRAINPUT, STREET of value "12 Maple St.          " input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_ADDRESS.STREET

26.  %TSS-I-TRAINPUT, CITY of value "Newport         " input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_ADDRESS.CITY

27.  %TSS-I-TRAINPUT, STATE of value "RI" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_ADDRESS.STATE

28.  %TSS-I-TRAINPUT, ZIP of value "02840" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_ADDRESS.ZIP

29.  %TSS-I-TRAINPUT, SEX of value "M" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.SEX

30.  %TSS-I-TRAINPUT, BIRTH_DATE of value "01Sep11" input to
     EMPLOYEE_RECORD.EMPLOYEE_RECORD.BIRTH_DATE

31.  %TSS-I-TRARET, TSS$REQUEST returned on  8-NOV-1986 11:23:30.91

32.  %TSS-I-TRACALL, TSS$REQUEST called on  8-NOV-1986 11:23:31.05

33.  %TSS-I-TRAPRMCHN, channel is 898528

34.  %TSS-I-TRAPRMREQNAM, request name is EMPLOYEE_MENU_REQUEST

35.  %TSS-I-TRAOUTPUT,
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.FIRST_MESSAGE_FIELD of value
     "                                  " output to FIRST_MESSAGE_FIELD

36.  %TSS-I-TRAOUTPUT,
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.SECOND_MESSAGE_FIELD of value
     "                                  " output to SECOND_MESSAGE_FIELD

37.  %TSS-I-TRAINPUT, SELECTION of value "5" input to
     EMPLOYEE_WORKSPACE.EMPLOYEE_WORKSPACE.SELECTION

38.  %TSS-I-TRAINPUT, EMPLOYEE_NUMBER of value "       "
     input to EMPLOYEE_RECORD.EMPLOYEE_RECORD.EMPLOYEE_NUMBER

39.  %TSS-I-TRARET, TSS$REQUEST returned on  8-NOV-1986 11:23:37.88

40.  %TSS-I-TRACALL, TSS$CLOSE_RLB called on  8-NOV-1986 11:23:37.97

41.  %TSS-I-TRARET, TSS$CLOSE_RLB returned on  8-NOV-1986 11:23:37.99

42.  %TSS-I-TRACALL, TSS$CLOSE called on  8-NOV-1986 11:23:38.00

43.  %TSS-I-TRARET, TSS$CLOSE returned on  8-NOV-1986 11:23:38.19
```

Figure 17-1: Sample Trace Output (Cont.)

Notes to Figure 17-1:

- Lines 1 through 5 indicate entry and exit times of TSS$TRACE_ON, TSS$OPEN_RLB, and TSS$OPEN calls.
- Lines 7, 11, 19, and 33 show the channel number passed on the TSS$REQUEST call. It is the same number that was assigned on the TSS$OPEN call.
- Lines 8, 12, 20, and 34 show the request name passed on the TSS$REQUEST call.
- Lines 6, 9, 10, 17, 18, 31, 32, and 39 indicate entry and exit times of the request calls.
- Lines 13 through 16, 21 through 30, and 35 through 38 show the data that was transferred between the form and the program record during each request.
- Lines 40 through 43 show the entry and exit times of TSS$CLOSE_RLB and TSS$CLOSE.

17.3 Debugging an Application Using Two Terminals

If you use the VAX Symbolic Debugger to debug your programs, you can use two terminals and the TDMS Trace facility to make your job easier. If you use one terminal and set breakpoints at specific lines, the output from the debugger appears over the form, making your job much more difficult.

One of the terminals should be logged in (term-1), and the second one should be logged out (term-2). Note that the second terminal must be logged out because you can assign a terminal to your process only if it is not in use. Also note that to assign another terminal you need the system privilege SYSPRV. On term-1 issue the following commands at DCL level:

```
$ DEFINE DBG$INPUT term-2
$ DEFINE DBG$OUTPUT term-2
$ DEFINE TSS$TRACE_OUTPUT term-2
```

At term-1 type:

```
$ RUN program-name
```

On term-2 you see:

```
DBG>
```

When you issue these commands:

- You can issue all debug commands at term-2
- You see all debugger output on term-2
- You see TDMS trace output on term-2

Using two terminals lets you keep the form intact, examine variables, and see the trace output on the second terminal. You can now enter all debugger commands on term-2 and run the application from term-1.

When the program exits, term-1 is returned to command level and term-2 is returned to its original state.

Application Function Keys (AFKS) 18

This chapter explains how to redefine terminal keys from the application program.

18.1 What Are Application Function Keys?

Application function keys (AFKs) are keys that trigger special application-specific actions. AFKs are not restricted to the functions TDMS provides.

When the operator presses a key that the application program defines as an AFK, either or both of the following events occurs:

- An event flag is set
- A user-written asynchronous system trap (AST) routine is invoked

If you are unfamiliar with AST routines, you should read the *VAX/VMS System Services Reference Manual* before continuing with this chapter.

18.2 When Do You Use Application Function Keys?

You should use AFKs when you want to associate a terminal key with an action that is unrelated to TDMS or that interrupts TDMS. For instance, you can write an AST routine that calls TSS$CANCEL and then declare an AFK that invokes that AST routine. This way, the operator can cancel a request without entering any data, even if the form defines the fields as Response Required.

18.3 Declaring Application Function Keys

You declare an AFK with the TSS$DECL_AFK call. The code to declare an AFK is as follows.

BASIC

```
Return_status = TSS$DECL_AFK ( Channel BY REF,        &
                               Key_id  BY REF,        &
                               Event_flag BY REF,     &
                               Ast_routine BY REF,    &
                               Ast_parameter BY VALUE)
```

COBOL

```
CALL "TSS$DECL_AFK"
     USING BY REFERENCE Channel,
           BY REFERENCE Key-id,
           BY REFERENCE Event-flag,
           BY REFERENCE Ast-routine,
           BY VALUE Ast-parameter,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$DECL_AFK(%REF(Channel),
1                            %REF(Key_id),
2                            %REF(Event_flag),
3                            %REF(Ast_routine),
4                            Ast_parameter)
```

Channel is the channel number that was assigned on the TSS$OPEN call.

Key-id is a code representing the AFK. When the operator presses the key represented by the key-id parameter, the event flag will be set and the AST routine will be invoked. These parameters are optional but you must include at least one. See Table 18-1 for a list of application function keys.

Event-flag is the event flag that is set when the operator presses the AFK. This parameter is optional; if it is not present, TDMS does not set an event flag when the operator presses the key. However, if you do not specify an event flag, you must specify an AST routine.

Ast-routine is a subroutine. This parameter is optional. When the operator presses the AFK, TDMS calls this routine at AST level. You may use either the event flag or the AST routine by itself, or together.

Ast-parameter is a parameter to be passed to the AST routine. This parameter is optional. If the AST parameter is not present and an AST routine is, TDMS will pass an AST parameter of zero. TDMS treats this parameter as a value: you can pass any type of parameter you would like your AST routine to receive, including addresses (parameters by reference).

18.3.1 Terminal Keys You Can Declare as AFKs

Table 18-1 lists the valid key codes and the keys they represent.

Table 18-1: Application Function Key Codes

Key Id	Control Key	Key Id	Control Key
0	CTRL/space bar	15	CTRL/O
1	CTRL/A	16	CTRL/P
2	CTRL/B	18	CTRL/R
3	CTRL/C	20	CTRL/T
4	CTRL/D	21	CTRL/U
5	CTRL/E	22	CTRL/V
6	CTRL/F	23	CTRL/W
7	CTRL/G	24	CTRL/X
8	CTRL/H	25	CTRL/Y
9	CTRL/I	26	CTRL/Z
10	CTRL/J	27	CTRL/[
11	CTRL/K	28	CTRL/backslash
12	CTRL/L	29	CTRL/]
13	CTRL/M	30	CTRL/~
14	CTRL/N	31	CTRL/?

18.3.2 How to Write an AST Routine

When the operator presses an AFK that has an AST routine associated with it, TDMS invokes the AST routine and passes it three parameters. You must make sure your AST routine receives the parameters correctly. The calling sequence is as follows:

```
Return-status = AST-routine ( AST-parameter by value,
                              channel by reference,
                              key-id by reference)
```

The values AST-parameter, channel, and key-id that TDMS passes to the AST routine are the same values specified in the TSS$DECL_AFK call.

18.4 Removing an AFK Key Definition

To remove a key definition declared in a TSS$DECL_AFK call, you use the TSS$UNDECL_AFK call. The code to remove a key definition is as follows.

BASIC

```
Return_status = TSS$UNDECL_AFK (Channel BY REF,   &
                                 Key_id BY REF)
```

COBOL

```
CALL "TSS$UNDECL_AFK"
     USING BY REFERENCE Channel,
           BY REFERENCE Key-id,
     GIVING Return-status.
```

FORTRAN

```
 Return_status = TSS$UNDECL_AFK(%REF(Channel),
1                               %REF(Key_id))
```

Channel is the channel number that was assigned on the TSS$OPEN call.

Key-id is the code representing an AFK that was previously declared in a TSS$DECL_AFK call.

Using Asynchronous Calls 19

This chapter describes the TDMS asynchronous programming calls. You should have a complete understanding of the TDMS synchronous calls before you try to use TDMS asynchronous calls. See Chapters 14 and 15 of this manual for information on the TDMS synchronous calls. For further information on asynchronous calls and AST routines, see the *VAX/VMS System Services Reference Manual.*

19.1 What Are Asynchronous Calls?

When you invoke an asynchronous call, TDMS initiates the operation and then returns control immediately to the application program. TDMS provides an asynchronous equivalent for most of the synchronous calls. The asynchronous calls include:

- TSS$CLOSE_A
- TSS$COPY_SCREEN_A
- TSS$DECL_AFK_A
- TSS$OPEN_A
- TSS$READ_MSG_LINE_A
- TSS$REQUEST_A
- TSS$UNDECL_AFK_A
- TSS$WRITE_BRKTHRU_A
- TSS$WRITE_MSG_LINE_A

19.2 When Do You Use Asynchronous Calls?

Since asynchronous calls return control to the application program immediately, they are very useful for performing multiple actions at the same time.

One occasion when you might want to use asynchronous calls is if your database is on a remote network node. You can invoke requests asynchronously with the TSS$REQUEST_A; when the operator finishes entering one record, the next request is called immediately and the operator can begin entering the next record without having to wait. While the operator fills in the second form, the application program can write the first record to the database. In this way, the time delay caused by the network connection is not apparent to the operator.

19.3 The General Format for Asynchronous Calls

TDMS asynchronous calls are identical to the synchronous calls, with the following exceptions:

- The name of the asynchronous call has the suffix _A
- The call has four additional parameters

The general syntax of an asynchronous call is:

```
return-status = TSS$xxx_A  (  channel by reference,
                              [ rsb by reference ] ,
                              [ event-flag by reference ],
                              [ ast-routine by reference ],
                              [ ast-parameter by value ],
                              [ call-specific] , ... )
```

Return-status is the standard VAX/VMS return status indicating the success or failure of the call. The return status for an asynchronous call, if successful, indicates only that the call was initiated, not that it was completed.

Rsb is the address of a longword to receive the completion status for the call. This parameter is optional. If the parameter is not present, it is passed as a 0. However, if you do not specify a completion status, there is no way of knowing whether the call completed successfully or not.

Event-flag is the number of the event flag set when the call completes. This parameter is optional. If the parameter is not present, TDMS does not set an event flag for this call.

Ast-routine is the routine TDMS invokes when the call completes. This parameter is optional. However, either the event flag parameter or the AST routine parameter must be present for the call, or the application program has no way of knowing when the call completes.

Ast-parameter is a parameter TDMS passes to the AST routine when the call completes. This parameter is optional. If the AST parameter is not present, and an AST routine is, TDMS passes an AST parameter of zero to the AST routine.

Call-specific is a parameter specific to the call. For example, the call-specific parameters for the TSS$REQUEST call are the request library id, the name of the request, and the records to pass to the request.

See the *VAX TDMS Reference Manual* for more information on call-specific parameters for each call.

Index

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

* (asterisk)
  See Asterisk (*)
@ (at sign)
  See @file-spec command
! (exclamation point)
  See Exclamation point (!)
- (hyphen)
  See Hyphen (-)
; (semicolon)
  See Semicolon (;)

A

AFK
  See Application function keys
%ALL syntax, 3-3
  arrays
    elements of, 7-8
    horizontally-indexed, 8-7, 8-8f
    indexed, 7-15
    multiple, 7-20
    scrolled, 7-14
    two-dimensional, 8-7
  Display Only form fields, 3-6
  entire forms, 3-4
  errors in, 5-3, 5-7
  example of, 3-5
  form and larger record, 3-6
  form and smaller record, 3-7
  group record fields, 3-21
  informational messages in, 5-3
  rules, 3-3, 4-2
  when not to use, 3-19
  when to use, 3-4
  with explicit syntax, 3-19
  with INPUT TO, 1-11
  with OUTPUT TO, 1-12
Ambiguous field names, 3-14
  making unique
    using group field names, 3-15
    using record names, 3-17
    using the WITH NAME modifier, 3-17
  qualifying, 3-14
ANYMATCH case value, 6-10
Application function keys (AFKs), 18-1, 18-3t
  AST routines, 18-2, 18-4
  deassigning, 18-4
  declaring, 18-2
  event flags, 18-2

when to use, 18-1
Application programs
canceling TDMS calls, 15-6
compiling, 14-9
data types, 16-1, 16-26t
debugging
sample, 17-3 to 17-6f
Trace facility, 17-1
using log files, 17-2
using two terminals, 17-6
VAX Symbolic Debugger, 17-6
with TDMS calls, 17-2
declaring records, 16-1
in BASIC programs, 16-2
in COBOL programs, 16-7, 16-8
in FORTRAN programs, 16-13, 16-14
linking, 14-9, 14-10
opening RLB files, 14-2
passing records to requests, 14-4, 14-5, 16-25
reading from the message line, 15-1
record definitions in, 14-5
sample
BASIC program, 14-10 to 14-12
COBOL program, 14-12 to 14-13
sequence of TDMS calls, 14-1
signaling errors, 14-7
testing return status, 14-6
using control values, 6-3
using DBMS, 16-20 to 16-22
using Rdb/VMS, 16-19, 16-20
Arguments for TDMS calls, 14-1
ARRAY clause (CDDL), 7-6
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
nesting, 8-2
Arrays, 7-1
%ALL syntax, 7-8, 7-14
horizontally-indexed, 8-7
two-dimensional, 8-5, 8-7
bounds
adjusting, 9-6
in BASIC programs, 16-6, 16-7
in COBOL programs, 16-11
in RDU, 7-7
control values, 9-1
adjsting bounds, 9-6
evaluating at run time, 9-2
example, 9-2f
one-dimensional, 9-7, 9-8f
rules for specifying, 9-4
declaring
in BASIC programs, 16-5, 16-5f, 16-7f
in COBOL programs, 16-11, 16-11f, 16-12f
in FORTRAN programs, 16-16, 16-17f
explicit syntax, 7-7
for database scrolled regions, 16-23
for input, 16-25
initializing, 16-25
loading, 16-25
form, 7-2
horizontally-indexed, 8-1, 8-1f
indexed, 7-2, 7-3f
mapping from multiple, 7-20
scrolled, 7-3, 7-4f
simple, 4-3f
horizontally-indexed, 8-1f, 8-4, 8-9
indexed, 7-12
multiple, 7-18
nested
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
one-dimensional, 7-2f
as control values, 9-7, 9-8f
partial, 7-17, 8-10f
records
group, 7-5
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
multiple, 7-18
one-dimensional, 7-4, 7-5f
structure, 4-2f, 7-4
two-dimensional, 8-2, 8-4

using ARRAY syntax, 7-6, 8-2
using OCCURS syntax, 7-6, 8-2
with database streams, 16-24
rules, 7-9
scrolled, 7-12, 7-15
displaying, 10-5, 10-6f, 10-7
horizontally-indexed, 8-1f
multiple, 7-20
structure of, 7-2f
subscripts, 7-1
See also Dependent names
ranges, 7-1
two-dimensional
%ALL syntax, 8-7
in BASIC programs, 16-7
in COBOL programs, 16-12
in FORTRAN programs, 16-18
rules, 8-4
work, 9-6
zero-based
as control values, 9-6
in BASIC programs, 16-6
in COBOL programs, 16-11
in RDU, 7-7
AST routines
calling sequence, 18-4
for application function keys, 18-2, 18-4
for asynchronous calls, 19-2
parameters
for application function keys, 18-2
for asynchronous calls, 19-2, 19-3
writing, 18-4
Asterisk (*) character in Rdb/VMS DML, 16-20
Asynchronous calls
See TDMS programming calls
Asynchronous System Trap routines
See AST routines
Attributes
field
Display Only, 3-6
dummy fields, 10-5
Must Fill, 3-23
with CHECK modifier, 11-6
with NO CHECK modifier, 11-8
video, 1-12
in conditional instructions, 6-12
with HARDCOPY key, 15-5
with inactive forms, 5-7

B

BACK SPACE key, 10-1, 11-1t
Base instructions
in conditional requests, 6-2, 6-4f
in requests, 1-3
multiple OUTPUT TO instructions, 5-7
with PRKs, 11-11
BASIC programs, 14-1
CASE statement, 16-5
compiling, 14-9, 16-2
data types, 16-26t
CDDL, 16-3
declaring
record variants, 16-5
records, 16-2, 16-2f, 16-4f, 16-5f, 16-7f
subscript variables, 16-6
%INCLUDE statement, 16-2
linking, 14-9
samples, 14-10 to 14-12
using DBMS, 16-21
using Rdb/VMS, 16-19
VARIANT statement, 16-5
Bell, ringing the terminal, 15-4
Binary structures
storing, 2-9, 2-10
after modification, 2-11
BLINK FIELD instruction, 1-13
BOLD FIELD instruction, 1-13
Bounds of arrays, 7-6
adjusting, 9-6
in BASIC programs, 16-6, 16-7
in COBOL programs, 16-11
in RDU, 7-7
BUILD LIBRARY command (RDU), 12-4

/LOG qualifier, 7-10
Novalidate mode, 2-8
Building request libraries, 12-4 to 12-5
errors, 12-5
mapping messages in, 7-10
/NOSTORE qualifier, 2-10
Novalidate mode, 2-8
BYTE data type, 16-26t

C

Calling sequence
asynchronous TDMS calls, 19-2
for AST routines, 18-4
TDMS synchronous calls, 13-2
Calls
See TDMS programming calls
Canceling TDMS calls, 15-6, 18-1
CASE statement (VAX BASIC), 16-5
Case values
ANYMATCH, 6-10
case insensitivity, 6-8
for error checking, 9-7
in conditional requests, 6-4f
with PRKs, 11-11
match instructions, 6-12
multiple control values, 6-12
NOMATCH, 6-9
specifying, 6-9
structure of, 6-2
with control arrays, 9-2
with nested control field instructions, 6-9
CDD
copying requests, 2-10
DBM$RECORDS directory, 16-20
DBM$SUBSCHEMAS directory, 16-20
default directory
defining CDD$DEFAULT, 2-2
displaying, 2-3
setting, 2-2, 2-3
deleting requests, 2-12
modifying requests, 2-11
naming conventions, 2-3 to 2-4
path names, 2-3, 16-2
DBMS databases, 16-20
for Rdb/VMS databases, 16-19
in BASIC programs, 16-2
in COBOL programs, 16-8
in FORTRAN programs, 16-13
RDB$RELATIONS directory, 16-19
record definitions
DBMS, 16-20
in BASIC programs, 16-2
in COBOL programs, 16-7
in FORTRAN programs, 16-13
Rdb/VMS, 16-19
storing
request library definitions, 12-1
requests, 2-2, 2-8
CDD$DEFAULT
defining, 2-2
in login command file, 2-2
in RDU, 2-3
in BASIC programs, 16-2
in COBOL programs, 16-8
in FORTRAN programs, 16-13
with CREATE REQUEST command, 2-5
with DBMS, 16-21
with Rdb/VMS databases, 16-19
CDDL
ARRAY clause, 7-6
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
nesting, 8-2
array definitions, 7-1, 7-7
COPY FROM statement, 16-23
data types, 16-26t
in BASIC programs, 16-3, 16-5
in FORTRAN programs, 16-15
TDMS, 4-6
DESCRIPTION IS statement
in FORTRAN programs, 16-13
OCCURS clause, 7-6
in BASIC programs, 16-5
in COBOL programs, 16-11

in FORTRAN programs, 16-16
nesting, 8-2
record definitions, 1-2, 14-5
record names, 14-5
STRUCTURE statement
in BASIC programs, 16-3, 16-5
in COBOL programs, 16-9, 16-11
in FORTRAN programs, 16-15
VARIANT keyword
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
VARIANTS statement
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
Changing
request library definitions, 12-3
requests, 2-11
Channels
closing terminal, 14-6
opening terminal, 14-3
CHARACTER data type, 16-26t
CHECK modifier
assigning NO CHECK, 11-8
at run time, 11-7
default, 11-6
function of, 11-6
CLEAR SCREEN instruction, 1-6
Clearing the screen, 1-6, 14-6
Closing
I/O channels, 14-6
request libraries, 14-5
COBOL programs, 14-1
CDD path names in, 16-8
compiling, 14-9, 16-8
/LIST/COPY_LIST qualifiers, 16-8
COPY statement, 16-8
data types, 16-26t
declaring records, 16-7, 16-8, 16-9f, 16-10f, 16-11f, 16-12f
linking, 14-9
sample, 14-12 to 14-13
Command files
EDT startup, 5-8
login, 2-1, 2-2
RDU
creating requests, 2-6
default file type, 2-6
startup, 2-3
RDU$EDIT, 5-9
TDMS$EDIT, 5-9
Comment characters
in BASIC programs, 16-5
in DESCRIPTION instruction, 1-6
in FORTRAN programs, 16-13
TDMS, 1-6
Common Data Definition Language
See CDDL
Common Data Dictionary
See CDD
COMP-1 data type, 16-26t
COMP-2 data type, 16-26t
Compatibility
of data types, 4-4, 4-7
for input, 4-8
of field length and size, 4-9
of field structures, 4-3
of input mappings, 4-8t
of output mappings, 4-9t
of scale factors, 4-5
of sign conditions, 4-9
program and request records, 14-5
Compiling
BASIC programs, 16-2
COBOL programs, 16-8
FORTRAN programs, 16-13
TDMS programs, 14-9
COMPLEX data type, 16-26t
Condition codes
levels, 14-6
returned by TDMS calls, 13-2
signaling, 14-7
testing, 14-6
Conditional instructions
case values
ANYMATCH, 6-10
NOMATCH, 6-9
specifying, 6-9

CONTROL FIELD IS, 6-4f
control values
arrays, 9-1, 9-2f
multiple, 6-12
DISPLAY FORM instructions, 6-11
evaluation of, 6-7f
run-time, 6-5
multiple, 6-8
nesting, 6-8
structure of, 6-2
USE FORM instructions, 6-11
Conditional requests, 6-1, 6-2
CONTROL FIELD IS, 6-4f
evaluation of, 6-7f
structure of, 6-3f
using PRKs to return values to control values, 11-10
when to use, 6-3
Control field arrays
illegal nesting of dependent ranges, 9-5, 9-5f
rules for specifying, 9-4
CONTROL FIELD IS instruction, 6-4f
case values
ANYMATCH, 6-10
NOMATCH, 6-9
specifying, 6-9
control values, 6-6
evaluation of, 6-7f
run-time, 6-5
match instructions, 6-12
multiple, 6-12
nesting, 6-8
Control fields
debugging, 17-1
returning a value using PRKs, 11-10
Control values
arrays, 9-1, 9-2f
dependent names, 9-1
dependent ranges, 9-1
evaluating at run time, 9-2
one-dimensional, 9-7
rules for specifying, 9-4
two-dimensional, 9-10
multiple, 6-12
record fields, 6-6
specifying, 6-6
using in the program, 6-3
workspace arrays, 9-6
workspace records, 6-6
CONTROL Z
See CTRL/Z
Controlling application flow with program request keys, 11-8
COPY LIBRARY command (RDU), 12-2
COPY REQUEST command (RDU), 2-10
COPY statement (VAX COBOL)
CDD path names in, 16-8
format of translated record definition, 16-8
general format, 16-8
Copying
from CDD to VMS file, 2-12
record definitions
into BASIC programs, 16-2
into COBOL programs, 16-7
into FORTRAN programs, 16-13
request library definitions, 12-2
requests, 2-10, 2-12
the active form, 15-4
Correcting errors, 2-7
CREATE LIBRARY command (RDU), 12-1
CREATE REQUEST command (RDU), 2-5
CDD$DEFAULT in, 2-5
errors in, 2-5
path names in, 2-5
syntax errors during, 5-2
with command file, 2-6
with database streams, 16-24
with text file, 2-6
Creating
forms, 1-2
records, 1-2
request libraries, 12-4 to 12-5

request library definitions, 12-1 to 12-2
requests, 2-5
command-file method, 2-6
from DCL level, 2-6
interactively, 2-5
text-file method, 2-6
CTRL/Z, 2-13

D

D_FLOATING COMPLEX data type, 16-26t
D_FLOATING data type, 4-7t, 16-26t
Data Manipulation Language (DML)
DBMS, 16-21
Rdb/VMS, 16-19
Data types
allowable picture characters, 4-5t
compatibility, 4-7
for input, 4-8
of field lengths and sizes, 4-9
of field sign conditions, 4-9
conversion chart, 16-26t
form fields, 4-4
programming language support, 16-1
record fields, 4-6
DATATRIEVE
array definitions, 7-7
record names, 14-5
DATE data type, 4-7t, 16-26t
DBG$INPUT logical name, 17-6
DBG$OUTPUT logical name, 17-6
DBM$RECORDS directory, 16-20
DBM$SUBSCHEMAS directory, 16-20
DBMS
DBM$RECORDS directory, 16-20
DBM$SUBSCHEMAS directory, 16-20
path names, 16-20
record definitions, 16-20
naming conventions, 16-20
work areas, 16-21
sample programs, 16-21
schema names, 16-20
subschema definitions, 16-20
using scrolled regions with, 16-22
DCL commands
BASIC, 14-9
COBOL, 14-9
DEFINE, 2-2, 17-2
FORTRAN, 14-9
LINK, 14-9
TYPE, 5-9
Deassigning application function keys, 18-4
Debugging TDMS programs, 17-1
sample, 17-3 to 17-6f
using log files, 17-2
using two terminals, 17-6
VAX Symbolic Debugger, 17-6
with TDMS calls, 17-2
DECIMAL data type, 16-26t
DECLARE statement (VAX BASIC), 16-2
Declaring
application function keys, 18-2
AST routines, 18-2, 18-4
event flags, 18-2
records, 16-1
explicitly, 16-1
in BASIC programs, 16-2, 16-2f, 16-4f, 16-5f, 16-7f
in COBOL programs, 16-7, 16-8, 16-9f, 16-10f, 16-11f, 16-12f
in FORTRAN programs, 16-13, 16-14, 16-14f, 16-15f, 16-17f
TDMS programming calls, 14-7
DEFAULT FIELD instruction
with inactive forms, 5-7
Defaults
AST parameters, 18-2
CDD directory, 2-2
showing, 2-3
dependent names, 9-2
error message level, 5-4
file types
for executable images, 14-10

for object files, 14-9
for RDU command files, 2-6
for request library files, 14-2
for the SAVE command, 5-9
form fields
data types, 4-4
scale factor, 4-5
I/O device, 14-3
RDU editor, 2-7, 5-9, 12-3
run-time function keys, 11-1t
VMS directory, 14-9
DEFINE command (DCL)
CDD$DEFAULT, 2-2
TSS$TRACE_OUTPUT, 17-2
Defining
CDD$DEFAULT, 2-2
forms, 1-2
RDU
default editor, 2-7
in login command files, 2-1
symbol, 2-1
RDU$EDIT, 5-9
records, 1-2
request libraries, 12-1
TSS$TRACE_OUTPUT, 17-2
DELETE command (DMU), 2-12
DELETE LIBRARY command (RDU), 12-4
DELETE REQUEST command (RDU), 2-12
Deleting
from the CDD, 2-12
request library definitions, 12-4
requests, 2-12
Dependent names, 9-1
%ENTRY, 9-2
redefining, 9-6
%LINE, 9-2
redefining, 9-6
Dependent ranges, 9-1
DESCRIPTION instruction, 1-6
semicolon in, 1-13
DESCRIPTION IS statement (CDDL)
in FORTRAN programs, 16-13
Dictionary Management Utility
See DMU commands
DICTIONARY statement (VAX FORTRAN)
general format, 16-13
/LIST qualifier, 16-13
Directories
CDD default, 2-2
VMS default, 5-9, 12-4, 14-9
DISPLAY FORM instruction, 1-6
errors in, 5-5, 5-7
given names in, 2-4
in conditional instructions, 6-11
WITH OFFSET modifier, 5-6, 5-7
Displaying
default CDD directory, 2-3
forms, 1-6
with an offset, 5-6
request library definitions, 12-2
requests, 2-11
scrolled regions, 10-5, 10-7
DML
See Data Manipulation Language
DMU commands
DELETE, 2-12
EXTRACT, 2-12
LIST, 4-12
Documenting requests, 1-6
DOUBLE data type, 16-26t
Down arrow key, 10-1, 11-2t

E

EDIT command (RDU), 2-7, 5-8
default editor, 5-9
Editing
RDU commands, 2-7
request library definitions, 12-3
requests, 2-11
Editor
changing, 5-9
default, 2-7, 5-8, 12-3
Elements of arrays, 7-1
explicitly mapping, 7-7
Enabling the Trace facility, 17-1

END DEFINITION instruction, 1-13
Ending
  request definitions, 1-13
  request instructions, 1-13
ENTER key, 11-2t
Entering RDU, 2-1
%ENTRY lexical function, 9-1, 9-2
  redefining, 9-6
Erasing the screen, 1-6, 14-6
Error level status, 14-6
Errors
  correcting, 5-8 to 5-10
    with the SAVE command, 5-9
  form-related, 5-7
  in mappings, 5-3, 5-7
    arrays, 7-10
    explicit, 5-4
  in request library definitions, 12-2
  in TDMS programs, 14-6
  notifying the operator
    with TSS$WRITE_BRKTHRU, 15-3
    with TSS$WRITE_MSG_LINE, 15-3
  semantic, 5-3
  severity, 5-2
  signaling, 14-7
  syntax, 5-1
    how RDU reports, 5-2
    in Interactive mode, 5-2
  tracing at run time, 17-1
  while building request libraries, 12-5
Evaluating conditional instructions, 6-5
Event flags
  for asynchronous calls, 19-2
  with application function keys, 18-2
Exclamation point (!) comment character, 1-6, 1-13
Executing a request, 14-3
Execution order
  errors in, 5-6
  of conditional instructions, 6-5
EXIT command (RDU), 2-13
Exiting RDU, 2-13
Explicit mapping
  arrays, 7-7, 7-12
  errors in, 5-3, 5-4
  group record fields, 3-21
  multiple fields, 3-10
  partial arrays, 7-17
  rules, 3-3, 4-1
    for specifying fields, 3-9
  syntax, 3-3
  when to use, 3-9
  with %ALL syntax, 3-19
EXTRACT command (DMU), 2-12

F

F12 key, 11-1t
F13 key, 11-1t
F15 key, 11-2t
F_FLOATING COMPLEX data type, 16-26t
F_FLOATING data type, 4-7t, 16-26t
Fatal level status, 14-6
FDU, 1-2, 4-11
FDU commands
  LIST FORM, 4-11
Field names
  conventions, 3-3
  making unique
    using group field names, 3-15
    using record names, 3-17
    using the WITH NAME qualifier, 3-17
  qualifying, 3-14
  record arrays, 7-4
Fields
  data types, 4-4
  form field structures, 4-3
  mapping, 1-9
    %ALL syntax, 1-11
    arrays, 7-6
    multiple fields, 3-10, 3-21
    restrictions, 3-23
    with %ALL syntax, 1-12
  naming conventions, 3-3

record structures, 4-2f, 4-3
video attributes, 1-12
FILE IS instruction
file name in, 12-1
in request library definition, 12-5
File method of creating requests, 2-6
@file-spec command (RDU), 2-6
Files
names
default for SAVE command, 5-9
request library files, 12-2, 12-4
request libraries, 12-1
TDMS shareable image, 14-9
TSS$HARDCOPY, 15-4
TSS$TRACE_OUTPUT, 17-2
Flags
See Event flags
Floating-point data types, 4-7t
Form arrays, 7-1, 7-2
horizontally-indexed, 8-1, 8-1f
indexed, 7-2, 7-3f
mapping, 7-6, 7-8
multiple, 7-20
with %ALL, 7-8
scrolled, 7-3, 7-4f
collecting data from, 10-2 to 10-5
displaying, 10-5, 10-6f, 10-7
Form Definition Utility
See FDU
Form definitions, 1-2
listing, 4-11
Form fields
data types, 4-4, 4-5t
length, 4-4
picture characters, 4-4
picture strings, 4-4
scale factors, 4-4
simple, 4-3f
structures, 4-3, 4-3f
array, 4-3
simple, 4-3
FORM IS instruction
error messages, 1-5
general format, 1-4
given names in, 2-4
in conditional requests, 6-12
path names in, 1-5, 2-4
with DISPLAY FORM, 5-5
WITH NAME modifier, 1-5, 5-6
with USE FORM, 5-5
Forms
active, 5-7
with ANYMATCH case value, 6-11
with NOMATCH case value, 6-10
CDD path names, 2-3
copying to a file, 15-4
defining, 1-2
displaying, 1-6
listing, 4-11
mapping
entire forms, 3-4
to larger record, 3-6
to smaller record, 3-7
offset on the screen, 5-6
unique names, 1-5
using in requests, 1-4
validating request references, 2-7
video attributes, 1-12
FORTRAN programs, 14-1
CDD path names in, 16-13
CDDL data types in, 16-15
comment character for CDDL descriptions, 16-13
compiling, 14-9, 16-13
contents of listing file, 16-13
data types, 16-26t
declaring
records, 16-13, 16-14, 16-14f, 16-15f, 16-17f
DICTIONARY statement
/LIST qualifier, 16-13
/G_FLOATING qualifier, 16-27
/I4 qualifier, 16-27
linking, 14-9
listing, 16-13
MAP statement, 16-16
two-dimensional arrays, 16-18
UNION statement, 16-16

Full path names, 2-3
  in RDU, 2-4
  in request library definitions, 12-2
Function keys
  application, 18-1, 18-3t
    deassigning, 18-4
    declaring, 18-2
    when to use, 18-1
  run-time, 11-1t
Functions
  See Lexical functions

G

G_FLOATING COMPLEX data type, 16-26t
G_FLOATING data type, 4-7t, 16-26t
GFLOAT data type, 16-26t
Given names, 2-3
  and logical names, 2-4
  in DISPLAY IS, 2-4
  in FORM IS, 2-4
  in RDU, 2-4
  in RECORD IS, 2-4
  in REQUEST IS, 2-4
  in USE FORM, 2-4
GOLD key, 11-2t
GOLD key in program request keys, 11-4, 11-5
Group record
  with %ALL, 3-21
Group record arrays, 7-5, 7-5f
  two-dimensional, 8-3f
Group record fields, 3-15, 4-2f
  mapping, 3-21
    restrictions, 3-23

H

H_FLOATING COMPLEX data type, 16-26t
H_FLOATING data type, 4-7t, 16-26t
HARDCOPY key, 11-2t, 15-4
  video attributes, 15-5
  when to use, 15-5
Header instructions
  in conditional requests, 6-2, 6-4f
  in requests, 1-3
HELP key, 11-2t
HFLOAT data type, 16-26t
Horizontally-indexed scrolled arrays, 8-1
  %ALL syntax, 8-7
  entire array, 8-5
  partial, 8-9
  rules, 8-4
  size of, 8-1
  syntax, 8-4
Hyphen (-)
  continuation character, 2-6
  in COBOL programs, 16-8

I

I/O channels
  closing terminal, 14-6
  opening terminal, 14-3
Implicit mapping
  See %ALL syntax
%INCLUDE statement (VAX BASIC)
  CDD path names, 16-2
  general syntax, 16-2
Including record definitions
  in BASIC programs, 16-2
  in COBOL programs, 16-7
  in FORTRAN programs, 16-13
Indexed form arrays, 7-2, 7-3f
  %ALL syntax, 7-8, 7-15, 7-16f
  mapping partial, 7-17
Informational level messages, 5-3
Informational level status, 14-6
Input mappings, 1-9
  %ALL syntax, 1-11
  compatibility, 4-8t
    of data types, 4-8
  from a scrolled region, 10-1
  from the message line, 15-1
  without operator input, 3-2
INPUT TO instruction, 1-9, 3-2
  %ALL syntax, 1-11
    See also %ALL syntax

commas in, 1-9
example of, 3-5
in conditional requests, 6-5
mapping arrays, 7-8, 7-15
with PRKs, 11-8
Instances of records
See Declaring records
Instructions
See Request instructions
INTEGER data type, 16-26t
with FORTRAN /I4 qualifier, 16-27
Interactive mode, 2-5
reporting syntax errors, 5-2
Interrupting requests, 15-3
Invoking
RDU, 2-1
requests, 14-3
Trace facility, 17-1

K

KEYPAD MODE IS instruction, 11-5
Keys
See also Program request keys
application function keys, 18-1, 18-3t
deassigning, 18-4
declaring, 18-2
when to use, 18-1
BACK SPACE, 10-1, 11-1t
CTRL/R, 11-2t
CTRL/W, 11-2t
down arrow, 10-1, 11-2t
ENTER, 11-2t
F12, 11-1t
F13, 11-1t
F15, 11-2t
GOLD, 11-2t
HARDCOPY, 11-2t, 15-4
HELP, 11-2t
left arrow, 11-2t
LINE FEED, 11-1t
PF2, 11-2t
PF4, 11-2t
RETURN, 11-2t
right arrow, 11-2t
run-time, 11-1t
TAB, 10-1, 11-1t
up arrow, 10-1, 11-2t
viewing scrolled regions, 10-1

L

Leaving RDU, 2-13
Left arrow key, 11-2t
LEFT OVERPUNCHED NUMERIC data type, 4-7t, 16-26t
LEFT SEPARATE NUMERIC data type, 4-7t, 16-26t
Length
of form fields, 4-4
of record fields, 4-6
Lexical functions
%ALL
example, 1-12
%ALL syntax, 1-12
See also %ALL syntax
%ENTRY, 9-1, 9-2
%LINE, 9-1, 9-2
Libraries
See Request libraries
Limits of an array, 7-6
LINE FEED key, 11-1t
%LINE lexical function, 9-1, 9-2
redefining, 9-6
LINK command (DCL), 14-9
Linking TDMS programs, 14-9, 14-10
LIST command (DMU), 4-12
LIST FORM command (FDU), 4-11
LIST LIBRARY command (RDU), 12-2
/LIST qualifier (VAX FORTRAN), 16-13
LIST REQUEST command (RDU), 2-11
general format of, 2-11
sample output, 2-12
Listing
BASIC programs with CDD records, 16-2

debugger log files, 17-2
form definitions, 4-11
FORTRAN programs with CDD records, 16-13
record definitions, 4-12
request library definitions, 12-2
requests, 2-11
/LOG qualifier
with %ALL syntax, 7-16
with BUILD LIBRARY command, 7-10
Logging errors, 17-2
Logical names
and CDD given names, 2-4
CDD$DEFAULT, 2-2, 2-3
DBG$INPUT, 17-6
DBG$OUTPUT, 17-6
RDU$EDIT, 5-9
TDMS$EDIT, 5-9
to enable Trace facility, 17-2
TSS$HARDCOPY, 15-4
TSS$TRACE_OUTPUT, 17-6
Login command files
defining CDD$DEFAULT, 2-2
defining RDU, 2-1
LONG data type, 16-26t

M

MAP statement (VAX BASIC), 16-2
MAP statement (VAX FORTRAN), 16-16
Mapping
%ALL syntax, 3-3
arrays, 7-8, 7-14, 7-15, 8-7
Display Only form fields, 3-6
entire forms, 3-4
errors in, 5-7
form and larger record, 3-6
form and smaller record, 3-7
when not to use, 3-19
when to use, 3-4
with explicit syntax, 3-19
with OUTPUT TO, 1-12
arrays, 7-6
%ALL syntax, 7-8, 7-14, 7-15, 8-7
explicit syntax, 7-12, 8-5
horizontally-indexed, 8-5
horizontally-indexed scrolled, 8-1
partial, 7-17
rules, 7-9
at run time, 14-3
compatibility
of data types, 4-4, 4-7
of field length and size, 4-9
of input, 4-8t
of output, 4-9t
of sign conditions, 4-9
errors, 5-3, 5-7
explicit syntax, 3-3
errors in, 5-4
specifying fields, 3-9
when to use, 3-9
with %ALL syntax, 3-19
group record fields, 3-21
restrictions, 3-23
multiple fields, 3-10, 3-21
restrictions, 3-13, 3-23
names
identical, 3-13
making unique, 3-13
rules, 3-3, 4-1
several form arrays, 7-20
several record arrays, 7-18
Mapping instructions, 1-9
commas in, 1-14
function of, 1-9
INPUT TO, 1-9, 3-2
OUTPUT TO, 1-11, 3-2
parentheses in, 1-14
RETURN TO, 3-2
validating, 12-5
Mapping tables
simplified input, 4-8t
simplified output, 4-9t
Match instructions, 6-12
Matching
any case value, 6-10
no case values, 6-9

Message line
  reading from, 15-1
  writing to, 15-3
MESSAGE LINE IS instruction
  in PRKs, 11-5
Modifiers to request instructions, 1-14
MODIFY LIBRARY command (RDU), 12-3
MODIFY REQUEST command (RDU), 2-11
  correcting errors with, 5-8
  default editor, 5-9
  path names in, 2-11
Modifying
  request library definitions, 12-3
  requests, 2-11
Multiple conditional instructions, 6-8
Multiple control values, 6-12
Multiple fields
  mapping, 3-10, 3-21
    restrictions, 3-23

N

Names
  CDD path names, 2-3
  field name conventions, 3-3
  of DATATRIEVE records, 14-5
  resolving ambiguous, 3-13
  unique
    for forms, 1-5
    for records, 1-5
Naming conventions, 2-3
  fields, 3-3
  record definitions
    DBMS, 16-20
    Rdb/VMS, 16-19
  request library definitions, 12-1
  request library files, 12-2, 12-4
Nesting
  arrays, 16-5
  conditional instructions
    CONTROL FIELD IS, 6-8
NOMATCH case value, 6-9
Nonprocedural instructions, 1-4
/NOSTORE qualifier, 2-9
  in Validate mode, 2-10
Notifying the operator
  with TSS$READ_MSG_LINE, 15-1
  with TSS$WRITE_BRKTHRU, 15-3
  with TSS$WRITE_MSG_LINE, 15-3
Novalidate mode
  effects of, 2-8, 2-9f
  setting, 2-8
  /STORE qualifier in, 2-9
Numeric data types, 4-7t

O

OCCURS clause (CDDL), 7-6
  in BASIC programs, 16-5
  in COBOL programs, 16-11
  in FORTRAN programs, 16-16
  nesting, 8-2
Offsetting the form on the screen, 5-6
One-dimensional arrays, 7-2f
  as control values, 9-7
  group, 7-5f
  mapping, 7-7
  record, 7-4, 7-5f
Opening
  log files, 17-2
  request libraries, 14-2
  terminal for I/O, 14-3
Operator
  canceling TDMS calls, 18-1
  input from the message line, 15-1
  notifying
    with TSS$WRITE_BRKTHRU, 15-3
    with TSS$WRITE_MSG_LINE, 15-3
Optional parameters for TDMS calls, 14-1
Order of execution of conditional instructions, 6-5

Order of TDMS programming calls, 14-1
Output mappings, 1-11
  %ALL syntax, 1-12
  compatible, 4-9t
  to the message line, 15-3
OUTPUT TO instruction, 1-11, 3-2
  %ALL syntax, 1-12
    See also %ALL syntax
  example of, 3-5
  in PRKs, 11-5
  mapping arrays, 7-8, 7-15

P

PACKED DECIMAL data type, 16-26t
PACKED NUMERIC data type, 16-26t
Parameters for TDMS calls, 14-1
Partial mapping of form arrays, 7-17
Partial path names
  See Relative path names
Path names, 2-3
  for DBMS databases, 16-20
  for Rdb/VMS databases, 16-19
  full, 2-3
  given, 2-3
  in RDU, 2-4
  relative, 2-3
PF2 key, 11-2t
PF4 (HARDCOPY) key, 15-4
PF4 key, 11-2t
Picture characters
  and data types, 4-4, 4-5
  function of, 4-4
  resulting data type, 4-5t
Picture strings
  assigning, 4-4
  function of, 3-3
Positioning the cursor in a scrolled region, 10-3
Primary TDMS calls, 14-1
PRKs
  See Program request keys
Procedure calling standard, 13-2
PROGRAM KEY IS instruction, 11-5
Program request keys
  CHECK modifier, 11-6
  controlling application flow, 11-8
  creating request that uses, 11-4
  defined by TDMS, 11-1t
  definition of, 11-2
  examples of, 11-8 to 11-11
  GOLD key as, 11-4
  in conditional instructions, 6-12
  names of, 11-4
  PROGRAM KEY IS instruction, 11-5
  returning values to control values, 11-10
  run-time execution of, 11-3
  WAIT instruction with, 1-12
  when to use, 11-4
  with conditional instructions, 11-4
Programming calls
  See TDMS programming calls
Programming languages, 14-1
Programs
  See also Application programs
  canceling TDMS calls, 15-6
  compiling, 14-9
  debugging, 17-1
    sample, 17-3 to 17-6f
    using log files, 17-2
    using two terminals, 17-6
    VAX Symbolic Debugger, 17-6
  general concepts, 13-1
  linking, 14-9, 14-10
  reading from the message line, 15-1
  signaling errors, 14-7
  testing return status, 14-6
Prompts
  on the message line, 15-2
  RDU>, 2-1
  RDUDFN>, 2-6

Q

Qualifying field names, 3-14

R

Ranges of subscripts, 9-1
RDB$RELATIONS directory, 16-19
Rdb/VMS
  database path names, 16-19
  DML
    example in BASIC, 16-20
    wildcard character in, 16-20
  record definitions, 16-19
    naming conventions, 16-19
  sample program, 16-19
  using scrolled regions with, 16-22
RDU, 1-2
  correcting errors, 2-7, 5-8
  creating
    request library definitions, 12-1 to 12-2
    requests, 2-5
  default editor, 2-7
  defining symbol for, 2-1
  exiting, 2-13
  form-related errors, 5-7
  invoking, 2-1
  startup command file, 2-3
RDU commands
  BUILD LIBRARY, 12-4
  COPY LIBRARY, 12-2
  COPY REQUEST, 2-10
  CREATE LIBRARY, 12-1
  CREATE REQUEST, 2-5
  DELETE LIBRARY, 12-4
  DELETE REQUEST, 2-12
  EDIT, 2-7, 5-8
  EXIT, 2-13
  LIST LIBRARY, 12-2
  LIST REQUEST, 2-11
  MODIFY LIBRARY, 12-3
  MODIFY REQUEST, 2-11, 5-8
  SET DEFAULT, 2-3
  SET VALIDATE, 2-8
  SHOW DEFAULT, 2-3
  VALIDATE LIBRARY, 12-4
RDU editor
  changing, 5-9
  default, 5-9, 12-3
RDU error messages
  for %ALL mappings, 5-3
  syntax errors, 5-2
RDU$EDIT logical name, 2-7, 5-9
  changing, 5-9
RDUINI.COM file, 2-3
Reading from the message line, 15-1
REAL data type, 16-26t
Record arrays, 7-1, 7-4
  ARRAY syntax, 7-6
  group, 7-5, 7-5f
  mapping, 7-6
    multiple, 7-18
  OCCURS syntax, 7-6
  one-dimensional, 7-4, 7-5f
  simple, 7-4
  two-dimensional, 8-2, 8-4
    ARRAY syntax, 8-2
    OCCURS syntax, 8-2
Record definitions, 1-2
  CDD path names, 2-3
  data types, 4-6
  DBMS, 16-20
    work areas, 16-21
  declaring, 16-1
    explicitly, 16-1
    in BASIC programs, 16-2, 16-2f, 16-4f, 16-5f, 16-7f
    in COBOL programs, 16-7, 16-8, 16-9f, 16-10f, 16-11f, 16-12f
    in FORTRAN programs, 16-13, 16-14, 16-14f, 16-15f, 16-17f
  defining workspaces, 6-6
  in BASIC programs, 16-2
  in COBOL programs, 16-8
  listing, 4-12
  making fields unique, 1-5, 3-17
    using group field names, 3-15
  Rdb/VMS, 16-19
  specifying in TDMS calls, 14-4
  using in requests, 1-5
  using the WITH NAME qualifier, 3-17
  validating request references, 2-7
  with VARIANTS

in BASIC, 16-5
Record fields
arrays
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
subscripts, 7-1
See also Dependent names
as control values, 6-6
data types, 4-6, 4-7t
length, 4-6
making unique, 3-13
group field names, 3-15
record names, 3-17
WITH NAME qualifier, 3-17
mapping multiple fields, 3-21
naming conventions, 3-13
qualifying, 3-14
scale factors, 4-6
structures, 4-2, 4-2f
array, 4-2
group, 4-2
simple, 4-2, 4-2f
RECORD IS instruction, 1-5
given names in, 2-4
in conditional requests, 6-12
mapping errors in, 5-7
path names in, 2-4
with DBMS path names, 16-21
WITH NAME modifier, 5-6
with workspace records, 6-6
RECORD statement (VAX FORTRAN), 16-13
Relations
See Rdb/VMS
Relative path names, 2-3
in RDU, 2-4
REPLACING phrase (VAX COBOL), 16-8
Reporting errors, 14-7, 19-2
Request base, 1-3
Request Definition Utility
See RDU
Request definitions
terminating, 1-13
Request header, 1-3
Request instructions, 1-3
CLEAR SCREEN, 1-6
DESCRIPTION, 1-6
DISPLAY FORM, 1-6
END DEFINITION, 1-13
ending, 1-13
FILE IS, 12-1, 12-5
FORM IS, 1-4
format, 1-4
INPUT TO, 1-9, 3-2
modifiers, 1-14
order of execution, 5-6
OUTPUT TO, 1-11, 3-2
PROGRAM KEY IS, 11-5
RECORD IS, 1-5
REQUEST IS, 12-1
RETURN TO, 3-2
rules for entering, 1-13
syntax rules, 1-13
USE FORM, 1-7
video field instructions, 1-12
WAIT, 1-12
with case values, 6-12
REQUEST IS instruction, 12-1
given names in, 2-4
path names in, 2-4
Request libraries, 12-1
building, 12-4 to 12-5
closing, 14-5
defining, 12-1
errors while building, 12-5
files, 12-1
opening, 14-2
validating, 12-5
Request library definitions, 12-1
building, 12-4
copying, 12-2
creating, 12-1 to 12-2
deleting, 12-4
errors in, 12-2
listing, 12-2
modifying, 12-3
naming conventions, 12-1
storing in the CDD, 12-1

validating, 12-3
Request library files
building, 12-4 to 12-5
closing, 14-5
default file type, 14-2
naming conventions, 12-2, 12-4
opening, 14-2
specifying, 12-5
in BUILD LIBRARY command, 12-4
in FILE IS instruction, 12-1
in RDU, 12-4
in TDMS programs, 14-2
Requests
base instructions, 1-3
binary structures, 2-9, 2-10, 2-11
building request libraries, 12-4
canceling, 15-6, 18-1
comments in, 1-6
concepts, 1-1
conditional, 6-1, 6-2
structure of, 6-3f
copying, 2-10
correcting errors, 2-7, 5-8 to 5-10
creating, 2-5
command-file method, 2-6
interactively, 2-5
text-file method, 2-6
data types, 4-7t
debugging, 17-1
deleting, 2-12
displaying a form, 1-6
ending, 1-13
executing, 14-3
format, 1-4f, 1-4
forms in, 1-4
header instructions, 1-3
interrupting, 15-3
listing, 2-11
mapping
arrays, 7-6
errors, 5-3
for input, 1-9
for output, 1-11
rules, 4-1
modifying, 2-11
naming conventions, 2-3 to 2-4
order of execution, 5-6
parts of, 1-3
passing program records to, 14-4, 14-5
records in, 1-5
request instructions, 1-3
specifying
in request libraries, 12-1
in TDMS calls, 14-4
storing, 2-2
syntax errors, 5-1
terminating, 1-14
using program request keys, 11-3
validating, 2-7, 2-10, 12-3, 12-5
RESET FIELD instruction, 1-13
with inactive forms, 5-7
RETURN key, 11-2t
Return status
for asynchronous calls, 19-2
levels, 14-6
signaling, 14-7
synchronous calls, 13-2
testing, 14-6
Return status block (rsb) parameter, 13-2, 19-2
RETURN TO instruction, 3-2
in PRKs, 11-5
mapping arrays, 7-8, 7-15
REVERSE FIELD instruction, 1-13
Right arrow key, 11-2t
RIGHT OVERPUNCHED NUMERIC data type, 4-7t, 16-26t
RIGHT SEPARATE NUMERIC data type, 4-7t, 16-26t
Ringing the terminal bell, 15-4
RLB files
See also Request library files
closing, 14-5
default file type, 14-2
specifying
in RDU, 12-4
in TDMS programs, 14-2

in the request library definition, 12-5
Rsb
See Return status block (rsb) parameter
Rules
for mapping arrays, 7-9
for specifying control value arrays, 9-4
Run-time evaluation
of conditional instructions, 6-5
of control value arrays, 9-2
Run-time function keys, 11-1t

S

SAVE command (RDU)
correcting errors with, 5-9
default file type, 5-9
viewing saved file, 5-9
Saving the current form, 15-4
Scale factor
default, 4-5
effect on data, 4-5t
of form fields, 4-4
of record fields, 4-6
Schema
See DBMS
Screen
attributes, 1-12
clearing, 1-6, 14-6
copying, 15-4
Scrolled form arrays, 7-2, 7-3, 7-4f
%ALL syntax, 7-14
collecting data from, 10-2 to 10-5
displaying, 10-1, 10-5, 10-7
horizontally-indexed, 8-1, 8-1f, 8-5, 8-7
mapping
%ALL syntax, 7-8
explicit syntax, 7-12
multiple, 7-20
partial, 7-17
to several record arrays, 7-18
windows, 7-3
Scrolled regions with database streams, 16-22
Semantic errors, 5-3
Semicolon (;)
errors, 5-2
terminating
comment text, 1-13
DESCRIPTION instruction, 1-6
request instructions, 1-13
SET DEFAULT command (RDU), 2-3
SET VALIDATE command (RDU), 2-8
Setting event flags, 18-2
Shareable images (TSSSHR.EXE), 14-9
SHOW DEFAULT command (RDU), 2-3
Signaling errors, 14-7
with TSS$WRITE_BRKTHRU, 15-3
with TSS$WRITE_MSG_LINE, 15-3
SIGNED BYTE data type, 4-7t, 16-26t
Signed data types, 4-7t
SIGNED LONGWORD data type, 4-7t, 16-26t
SIGNED NUMERIC data type, 16-26t
SIGNED OCTAWORD data type, 16-26t
SIGNED QUADWORD data type, 4-7t, 16-26t
SIGNED WORD data type, 4-7t, 16-26t
SINGLE data type, 16-26t
Special function keys, 11-1t
Startup files
EDT, 5-8
in RDU, 2-3
Status
asynchronous calls, 13-2
for asynchronous calls, 19-2
levels, 14-6
signaling, 14-7

synchronous calls, 13-2
testing TDMS calls, 14-6
Store mode
effects of, 2-9
/STORE qualifier, 2-9
in Novalidate mode, 2-9
Storing
form images, 15-4
request library definitions, 12-1
requests, 2-8, 2-9
STRING data type, 16-26t
STRUCTURE statement (CDDL)
in BASIC programs, 16-3, 16-5
in COBOL programs, 16-9, 16-11
in FORTRAN programs, 16-15
Structures
compatibility, 4-3
form field, 4-3, 4-3f
record, 4-2, 4-2f
Subfields, 7-5
Subschema
See DBMS
Subscripts, 7-1
%ALL syntax, 8-7
arrays used as control values, 9-6
dependent ranges, 9-1
explicit syntax, 8-9
in BASIC programs, 16-6
in FORTRAN programs, 16-17
in RDU, 7-7
in two-dimensional arrays, 8-2
limits, 7-6
%LINE and %ENTRY, 9-2
assigning values to, 9-6
ranges, 7-1
See also Dependent names
specifying in array mappings, 7-7
with database streams, 16-25
Success level status, 14-6
Symbolic Debugger, 17-6
DBG$INPUT logical name, 17-6
DBG$OUTPUT logical name, 17-6
Synchronous calls
See TDMS programming calls
Syntax errors, 5-1
correcting, 2-7
how RDU reports, 5-2
in Interactive mode, 5-2
Syntax rules for request instructions, 1-13
SYS$INPUT as default terminal, 14-3

T

TAB key, 11-1t
viewing scrolled regions, 10-1
Tables
See Arrays
TDMS data types, 16-26t
See Data types
TDMS programming calls
asynchronous, 13-2, 19-1
AST parameters, 19-2, 19-3
AST routines, 19-2
event flags, 19-2
parameters, 19-2 to 19-3
status block, 19-2
syntax, 19-2
when to use, 19-2
canceling, 15-6, 18-1
compiling, 14-9
debugging, 17-1
declaring, 14-7
errors in records, 5-7
format, 13-2
general concepts, 13-1
linking, 14-9, 14-10
return status, 13-2
signaling, 14-7
testing, 14-6
samples, 14-10 to 14-13
sequence, 14-1
synchronous, 13-2
TSS$CANCEL, 15-6
TSS$CLOSE, 14-6
TSS$CLOSE_RLB, 14-5
TSS$COPY_SCREEN, 15-4
TSS$DECL_AFK, 18-2
TSS$OPEN, 14-3
TSS$OPEN_RLB, 14-2

TSS$READ_MSG_LINE, 15-1
TSS$REQUEST, 14-3
TSS$SIGNAL, 14-7
TSS$TRACE_OFF, 17-2
TSS$TRACE_ON, 17-2
TSS$UNDECL_AFK, 18-4
TSS$WRITE_BRKTHRU, 15-3
TSS$WRITE_MSG_LINE, 15-3
using DBMS, 16-20 to 16-22
using Rdb/VMS, 16-19, 16-20
TDMS programs
See Application programs
TDMS shareable image, 14-9
TDMS$EDIT logical name, 5-9
Terminals
clearing the screen, 1-6, 14-6
closing, 14-6
opening, 14-3
ringing the bell, 15-4
specifying in TSS$OPEN, 14-3
using two to debug TDMS programs, 17-6
Terminating
request definitions, 1-13
request instructions, 1-13
TEXT data type, 4-7t, 16-26t
Text editor, RDU default, 2-7
Text-file method of creating requests, 2-6
Trace facility, 17-1
disabling
with TSS$TRACE_OFF, 17-2
enabling, 17-1
with logical names, 17-2
with TSS$TRACE_ON, 17-2
sample, 17-3 to 17-6f
TSS$_NORMAL status code, 14-7
TSS$CANCEL programming call, 15-6
parameter, 15-7
results of, 15-7
samples, 15-7
TSS$CLOSE programming call, 14-6
parameters, 14-6
samples, 14-6
TSS$CLOSE_RLB programming call, 14-5
parameters, 14-5
samples, 14-5
TSS$COPY_SCREEN programming call, 15-4
hardcopy function, 15-4
parameters, 15-6
samples, 15-5
when to use, 15-5
TSS$DECL_AFK programming call, 18-2
parameters, 18-2
samples, 18-2
TSS$HARDCOPY logical name, 11-2t, 15-4, 15-6
TSS$OPEN programming call, 14-3
parameters, 14-3
samples, 14-3
TSS$OPEN_RLB programming call, 14-2
parameters, 14-2
samples, 14-2
TSS$READ_MSG_LINE programming call, 15-1
parameters, 15-2
samples, 15-2
TSS$REQUEST programming call, 14-3
parameters, 14-4 to 14-5
samples, 14-4
TSS$SIGNAL programming call, 14-7
sample, 14-7, 14-8
TSS$TRACE_OFF programming call, 17-2
TSS$TRACE_ON programming call, 17-2
samples, 17-2
TSS$TRACE_OUTPUT logical name, 17-2, 17-6
TSS$UNDECL_AFK programming call, 18-4
parameters, 18-4
samples, 18-4

TSS$WRITE_BRKTHRU programming call, 15-3
parameters, 15-4
samples, 15-3
TSS$WRITE_MSG_LINE programming call, 15-3
parameters, 15-3
samples, 15-3
TSSSHR.EXE file, 14-9
Two-dimensional arrays, 8-2
%ALL syntax, 8-7
as control values, 9-10
group, 8-2
in BASIC programs, 16-7
in COBOL programs, 16-12
in FORTRAN programs, 16-18
mapping, 8-4
partial, 8-10f
rules, 8-4

U

UNDERLINE FIELD instruction
general format, 1-12
Underlying virtual array
See Scrolled form arrays
Underscore (_)
in COBOL programs, 16-8
UNION statement (VAX FORTRAN), 16-16
Unique names
for forms, 1-5
for records, 1-5
UNSIGNED BYTE data type, 4-7t, 16-26t
Unsigned data types, 4-7t
UNSIGNED LONGWORD data type, 4-7t, 16-26t
UNSIGNED NUMERIC data type, 4-7t, 16-26t
UNSIGNED OCTAWORD data type, 16-26t
UNSIGNED QUADWORD data type, 4-7t, 16-26t
UNSIGNED WORD data type, 4-7t, 16-26t
Up arrow key, 10-1, 11-2t
USE FORM instruction, 1-7
errors in, 5-5
form-related errors, 5-8
given names in, 2-4
in conditional instructions, 6-11
WITH OFFSET modifier, 5-6

V

VALIDATE LIBRARY command (RDU), 2-8, 12-4
Validate mode, 2-7
effects of, 2-9f
error checking, 5-3
/NOSTORE qualifier with, 2-10
VALIDATE REQUEST command (RDU), 2-8
Validating
mapping instructions, 12-5
request libraries, 2-8, 12-5
request library definitions, 12-3
requests, 2-7, 12-3
errors during, 5-3
VARIANT keyword (CDDL)
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
VARIANTS statement (CDDL)
in BASIC programs, 16-5
in COBOL programs, 16-11
in FORTRAN programs, 16-16
VARYING TEXT data type, 4-7t
VAX data types, 16-26t
VAX Procedure Calling Standard, 13-2
VAX Symbolic Debugger, 17-6
DBG$INPUT logical name, 17-6
DBG$OUTPUT logical name, 17-6
Video attributes
in conditional instructions, 6-12
instructions to control, 1-12
with HARDCOPY key, 15-5
with inactive forms, 5-7
Viewing scrolled regions, 10-1, 10-5, 10-7

Virtual array
  See Scrolled form arrays

W

WAIT instruction
  general format, 1-12
  with program request keys, 1-12
Warning level messages, 5-3
Wildcard character (*) in Rdb/VMS
    DML, 16-20
Windows, 7-3
  collecting data from, 10-2 to 10-5
  displaying data in, 10-5
  displaying scrolled regions, 10-1
WITH NAME modifier, 1-5
  making record names unique, 3-17
  uniqueness of names, 5-6
WITH OFFSET modifier, 5-6, 5-7
WORD data type, 16-26t
Work areas for DBMS, 16-21
Workspace records, 6-6
  arrays as control values, 9-6
Writing
  AST routines, 18-4
  messages
    to the trace file, 17-3
  to the message line, 15-3

Z

ZONED NUMERIC data type, 16-26t

How to Order Additional Documentation

If you live in:	Call:	or Write:
New Hampshire, Alaska	603-884-6660	Digital Equipment Corp. P.O. Box CS2008 Nashua, NH 03061-2698
Continental USA, Puerto Rico, Hawaii	1-800-258-1710	Same as above.
Canada (Ottawa-Hull)	613-234-7726	Digital Equipment Corp. 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: P&SG Business Manager or approved distributor
Canada (British Columbia)	1-800-267-6146	Same as above.
Canada (All other)	112-800-267-6146	Same as above.
All other areas	—	Digital Equipment Corp. Peripherals & Supplies Centers P&SG Business Manager c/o DIGITAL's local subsidiary

Note: Place prepaid orders from Puerto Rico with the local DIGITAL subsidiary (phone 809-754-7575).

Place internal orders with the Software Distribution Center, Digital Drive, Westminster, MA 01473-0471.

VAX TDMS
Request and
Programming Manual
AA-GS14B-TE

Reader's Comments

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

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

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

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

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

Name ______________________________ Date ______________

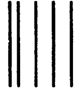
Organization ______________________________

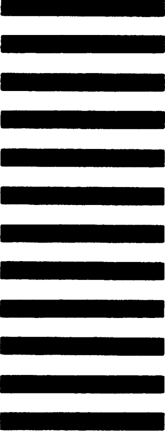
Street ______________________________

City ______________________________ State ______ Zip Code or Country ______________

------Do Not Tear - Fold Here and Tape --

digital





BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD MASS

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: DISG Documentation
ZK02-2/N53
Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062-2698

------Do Not Tear - Fold Here and Tape --

Cut Along Dotted Line