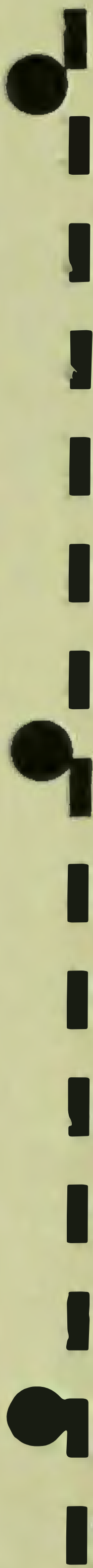
VAX DAL

Author's Guide

digital

software



94-003/049/02

VAX DAL Author's Guide

Order Number AA-K763C-TE

December 1985

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

First Printing, July 1982
Second Printing, July 1984
Third Printing, December 1985

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 error 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 © 1982, 1984, 1985 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:

DAL	DIBOL	ReGIS
DEC	Edusystem	Rainbow
DECtalk	GIGI	RSTS
DECmate	MASSBUS	RSX
DECnet	PDP	UNIBUS
DECUS	PDT	VAX
DECwriter	PROFESSIONAL	VMS
digital™		VT

C ID #77337

Contents

Contents

Preface

RELATED DOCUMENTS xiv

REFERENCES xiv

CONVENTIONS USED IN THIS MANUAL xv

Chapter 1
Introduction

THE DIGITAL AUTHORING LANGUAGE 1-2

- Graphics 1-2
- Judging The Student's Response 1-4
- Structure Of A Lesson 1-4

Chapter 2
Elements of VAX DAL

LESSON 2-1

INSTRUCTION 2-2

ARGUMENTS 2-3

- Keywords 2-3
- User-Defined Variables 2-3
- Data Types 2-4
- Usage Characteristics 2-5

- System Variables 2-6
- Constants 2-7
- Functions 2-8
- Expressions 2-8
- Default 2-9

SELECTING NAMES 2-9

SYNTAX 2-11

- Dot Indentation 2-12

USING THE ELEMENTS OF VAX DAL 2-12

Chapter 3
Writing on the Screen

SCREEN ADDRESSES 3-1

- Row-and-Column Coordinates 3-2
- Fine Coordinates 3-4
- Normalized Coordinates 3-8
- Screen Address Summary 3-10

CURRENT ATTRIBUTES 3-11

- Current Location 3-13

DISPLAYING BLOCKS OF TEXT 3-14

DISPLAYING VARIABLES 3-15

TEXT SIZE 3-18

USING COLOR 3-23

DISPLAY MODES 3-24

Chapter 4
Default Response Judging

THE RESPONSE-JUDGING BLOCK 4-1

- QUERY 4-2
- RIGHT And WRONG 4-4

DISPLAYING FEEDBACK 4-5

SPECIFICATIONS FOR MATCHING 4-7

Chapter 5
Creating a Simple Lesson

STRUCTURE OF A LESSON	5-2
• Planning The Lesson	5-2
• Planning Units	5-4
LESSON-LEVEL INSTRUCTIONS	5-4
• Lesson-Level Variables	5-5
• Lesson-Level Control Logic	5-6
DISPLAYING TEXT	5-9
• Changing Character Sizes	5-9
• Changing The System Prompt Character	5-13
• Using Color	5-15
• Changing Display Modes	5-15
• Displaying Variables	5-16
JUDGING THE STUDENTS' RESPONSES	5-17
• Specifying Several Answers	5-18
• Specifying Variables As Answers	5-19
• Modifying Response Judging	5-21
SCORING AND GOALS	5-22
CONTROL LOGIC	5-24
• The LOOP,ENDLOOP Structure	5-25
• The FOR,ENDFOR Structure	5-27
• The IF,ENDIF Structure	5-29
• The TEST,VALUE,ENDTEST Structure	5-34
• Combining Structures	5-36
ENDING THE LESSON	5-40

Chapter 6
Editing, Compiling, and Linking a Lesson

PREPARING A LESSON	6-1
• Testing And Correcting A Lesson	6-2
• Compiler Errors	6-3
• Run-time Errors	6-5

- Logic Errors 6-6

THE VAX DAL COMPILER 6-7

- Compiler Switches 6-7
- Compiler Examples 6-8

THE VAX VMS LINKER 6-9

Chapter 7
Terminal Management

DIFFERENCES BETWEEN TERMINAL MODELS 7-2

- Software Support Requirements 7-2
- Screen Characteristics 7-3
- Color Capabilites 7-4

THE VAX DAL TERMINAL MANAGEMENT INSTRUCTIONS 7-6

- Response Display Modifiers 7-6
- DAL Color System Modifiers 7-7
- The SET MAXCOLORS Instruction 7-7
- The SET HLS Instruction 7-8
- Instructions That Enable Special Function Keys 7-8
- The SET FKEY Instruction 7-8
- The SET KEYPAD Instruction 7-10
- The SET DELETE Instruction 7-10
- Instructions That Save And Restore Terminal States 7-11
- The SAVE Instruction 7-12
- The RESTORE Instruction 7-13

Chapter 8
Response Judging

MODIFYING WHAT A RESPONSE MATCHES — THE SPECS INSTRUCTION 8-1

- Arguments To The SPECS Instruction 8-2
- Using The SPECS Instruction — ANYORDER And EXTRA Keywords 8-4

MODIFYING WHAT A RESPONSE MATCHES — THE SYN INSTRUCTION 8-4

MODIFYING WHAT THE RESPONSE MATCHES —
THE NOISE INSTRUCTION 8-5

MODIFYING JUDGMENT OF THE RESPONSE — THE JUDGE INSTRUCTION 8-6

- Arguments To The JUDGE Instruction 8-6
- Using The JUDGE Instruction — the CONTINUE Keyword 8-7
- Using The JUDGE Instruction — the STOP Keyword 8-8

THE RIGHTV AND WRONGV INSTRUCTIONS 8-8

- Student Variables 8-9
- Specifying A Tolerance 8-11
- Specifying Units 8-12

MISCELLANEOUS DISPLAY INSTRUCTIONS 8-14

- The PROMPT Instruction 8-14
- The INPUT Instruction 8-15

Chapter 9
Response Processing

SCORING SYSTEM VARIABLES 9-1

RESPONSE-RELATED SYSTEM VARIABLES 9-5

USING SYSTEM VARIABLES 9-9

SYSTEM VARIABLES IN NESTED QUERIES 9-11

USING THE ERRORV SYSTEM VARIABLE 9-14

- Setting ERRORV Values 9-15
- Checking ERRORV Values In A Response-Judging Block 9-16
- Interpreting ERRORV Values 9-18
- Interpreting ERRORV Values For Expressions 9-18
- Interpreting ERRORV Values For String Responses 9-20

Chapter 10
Graphics

BASIC GRAPHICS INSTRUCTIONS 10-2

- BOX 10-4
- CIRCLE 10-5
- CURVE 10-7
- DOT 10-10
- LINE 10-10
- VECTOR 10-10

MODIFYING LINE GRAPHICS 10-12

- PATTERN 10-12
- SREF 10-13

TEXT ENHANCEMENT 10-16

- ITALICS 10-16
- TROTATE 10-17

ERASE 10-19

MODE 10-20

RELATIVE GRAPHICS 10-23

- RORIGIN 10-24
- RSIZE 10-26
- ROTATE 10-28

GRAPHICS SYSTEM VARIABLES 10-31

GENERAL GRAPHICS CONSIDERATIONS 10-36

- Using Color 10-37
- Terminal Line Speed 10-37

Chapter 11
Modifying Lesson Flow

THE UNIT CALLING CHAIN 11-2

UNCONDITIONAL TRANSFER OF CONTROL 11-3

- The RETURN Instruction 11-3
- The BACKUP Instruction 11-3

CONDITIONAL TRANSFER OF CONTROL 11-8

- The BRANCH Instruction 11-8
- The WHEN Instruction 11-10
- Condition Handlers 11-16
- The ON Instruction 11-16
- The CDUNIT Instruction 11-19
- The SIGNAL Instruction 11-20
- The CANCEL Instruction 11-20
- A Condition-Handling Example 11-21

Chapter 12
The VAX DAL Color Management System

THE COMPONENTS OF THE DAL COLOR MANAGEMENT SYSTEM 12-2

- Color Specifications 12-2
- Terminal Color Palettes 12-3
- The DAL Color Map 12-3
- The DAL Color Table 12-5

THE VAX DAL COLOR MANAGEMENT INSTRUCTIONS 12-7

- The FCOLOR Instruction 12-7
- The CCOLOR Instruction 12-8
- The BCOLOR Instruction 12-9
- The MAP Instruction 12-10
- The SET MAXCOLORS Instruction 12-11

A COLOR MANAGEMENT EXAMPLE 12-11

Chapter 13
File Input/Output in VAX DAL

DATA ELEMENTS 13-2

- Records 13-2
- Data Fields 13-2
- Bytes 13-2

FILE STRUCTURES 13-3

- Sequential File Structure 13-3
- Access To Records In Sequential Files 13-3
- Random File Structure 13-4
- Access To Records In Random Files 13-4
- Indexed File Structure 13-5
- Access By Key Value To Records In Indexed Files 13-5

THE VAX DAL FILE I/O INSTRUCTIONS 13-6

- The OPEN Instruction 13-6
- The GET Instruction 13-7
- The PUT Instruction 13-7
- The FIND Instruction 13-7

- The UPDATE Instruction 13-7
- The DELETE Instruction 13-8
- The CLOSE Instruction 13-8

FILE I/O OPERATIONS WITH SEQUENTIAL FILES 13-8

- Reading Records From A Sequential File 13-9
- Writing Records To A Sequential File 13-10

FILE I/O OPERATIONS WITH RANDOM FILES 13-11

- Reading Records From A Random File 13-12
- Writing Records To A Random File 13-13

FILE I/O OPERATIONS WITH INDEXED FILES 13-15

- Reading Records From An Indexed File 13-17
- Writing Records To An Indexed File 13-19
- Updating Records In An Indexed File 13-20
- Deleting Records In An Indexed File 13-24

Chapter 14
Parameter Passing in VAX DAL

PASSING DATA TO DAL ROUTINES 14-1

PASSING DATA FROM DAL ROUTINES 14-2

PARAMETER-PASSING EXAMPLES 14-4

Chapter 15
Macros in VAX DAL

DEFINING A VAX DAL MACRO 15-2

USING PARAMETERS WITH A VAX DAL MACRO 15-3

USING THE %INCLUDE INSTRUCTION 15-4

Appendix A
Instructions

Appendix B
System Functions

Appendix C
System Variables

Appendix D
Operators

Appendix E
Syntax Symbols

Appendix F
Sample Lessons

MULTIPLY F-1

ICECREAM F-7

MENU_DRIVER F-11

Glossary

Index

Figures

1-1	Text Displays	1-3
1-2	Traffic Lesson	1-3
3-1	Row-and-Column Addresses	3-3
3-2	Fine Screen Addresses	3-6
3-3	Normalized Screen Coordinates	3-9
3-4	Current Location, Size, and Mode	3-12
3-5	Displaying Variables	3-16
3-6	Text Sizes	3-19
3-7	Changing Text Size	3-22
4-1	The QUERY Instruction	4-3
4-2	Response-Judging Block	4-4
4-3	Markup	4-6
5-1	Lesson Structure	5-3
5-2	Lesson-level Control Logic	5-8
5-3	The Unit Menu	5-10
5-4	The Unit Review	5-13
5-5	Changing the Prompt Character	5-14
5-6	Displaying Variables	5-16
5-7	Problem Judged Wrong	5-22
5-8	The FOR Instruction	5-29
5-9	The Unit Shoscore	5-31
5-10	Nested IF Instructions	5-32

5-11	The TEST Instruction	5-35
5-12	The Unit Instruction	5-38
5-13	The Example Problem	5-39
8-1	The RIGHTV Instruction	8-10
8-2	The CONVERT Instruction	8-14
9-1	Using ERRORV to Detect Missing Parentheses	9-17
9-2	Using ERRORV to Detect a Misspelled System Function	9-18
10-1	Basic Graphics Instructions: Sailboat	10-3
10-2	The BOX Instruction	10-5
10-3	The CIRCLE Instruction	10-6
10-4	The CURVE Instruction: Three Points	10-8
10-5	The CURVE Instruction: Four Points	10-9
10-6	The VECTOR Instruction	10-11
10-7	The PATTERN Instruction	10-12
10-8	Sailboat with Shaded Sail	10-14
10-9	The SREF Instruction	10-15
10-10	The SREF and PATTERN Instructions	10-16
10-11	The ITALICS Instruction	10-17
10-12	The TROTATE Instruction	10-18
10-13	The MODE Instruction 1	10-22
10-14	The MODE Instruction 2	10-22
10-15	The RORIGIN Instruction	10-25
10-16	The RSIZE Instruction	10-27
10-17	The ROTATE Instruction	10-29
10-18	Rotated Boxes: RBOX	10-30
10-19	Rotated Boxes: RLINE	10-31
10-20	Graphics System Variables 1	10-35
10-21	Graphics System Variables 2	10-36
11-1	A Unit Calling Chain	11-2
12-1	Default Color System: Lesson Startup	12-12
12-2	Modified Color Map	12-13

12-3 Initial Color Table Configuration 12-15
12-4 Final Color Table Configuration 12-17
13-1 Sequential File Organization 13-4
13-2 Random File Organization 13-5

Tables

3-1 Horizontal Fine Address Units 3-5
3-2 Default Character Heights 3-20
3-3 Character Dimensions in Fine Address Units 3-21
3-4 Mode Keywords 3-25
3-5 Mode System Constants 3-26
7-1 VAX DAL Software Support Requirements 7-2
7-2 Screen Widths 7-4
7-3 Terminal Color Capabilities 7-5
7-4 ASCII Strings Generated by Special Function Keys 7-9
7-5 ASCII Strings Generated by Keypad Keys 7-10
7-6 Terminal Characteristics Saved by the SAVE Instruction 7-12
9-1 Scoring System Variables 9-2
9-2 Response-Related System Variables 9-6
9-3 Items Saved when Queries are Nested 9-13
9-4 ERRORV Values 9-14
10-1 Graphics System Variables 10-32
11-1 Time-Related WHEN Keywords 11-11
11-2 String-Related WHEN Keywords 11-11
11-3 Condition-Name Keywords 11-17
11-4 IORESULT values 11-18
12-1 DAL-Provided Color Specifications 12-4
12-2 Terminal Models: Color Table Size 12-6
14-1 Default Parameter-Passing Methods 14-2
14-2 Parameter-Passing Options (Passing Data To External Routines Only) 14-4
C-1 Graphics and Graphing System Variables C-1
C-2 Response-Related System Variables C-4

C-3	Scoring System Variables	C-6
C-4	Timing and Miscellaneous System Variables	C-8
D-1	Operators	D-1
E-1	Syntax Symbols	E-1

Preface

This manual is an introduction to the DIGITAL Authoring Language (VAX DAL or DAL) and is intended for new authors. It assumes little knowledge of programming and explains some programming terms and concepts. The manual also introduces a subset of VAX DAL. After this introduction, the new author should be able to consult the *VAX DAL Reference Manual* for information about all instructions.

Experienced authors and programmers may find Chapters 7 through 15 helpful. These chapters deal with advanced subjects in VAX DAL.

Chapter 1 describes the authoring process and the capabilities of VAX DAL.

Chapters 2, 3, and 4 explain concepts that apply to writing all lessons. Chapter 2 describes the elements and syntax of the language. Chapter 3 discusses basic concepts and instructions for displaying text, including screen addresses, text sizes, and color. Chapter 4 discusses response judging and the sequence of events that occurs during response judging. It explains the basic instructions for reading the student's response from the keyboard and then judging it.

Chapter 5 illustrates the concepts and instructions discussed in Chapters 2, 3, and 4 by explaining a short lesson in detail. Chapter 5 also introduces the concept of control logic and the related VAX DAL instructions.

Chapter 6 describes how the VAX DAL compiler and VAX/VMS linker are used to prepare a lesson for execution.

Chapter 7 discusses terminal management: how authors can tailor terminal characteristics for a lesson. Chapters 8 and 9 deal with modifications to default response judging and processing. Chapter 10 describes the VAX DAL graphics instructions. Chapter 11 discusses advanced control logic and lesson flow. Chapter 12 describes the components and function of the DAL color management system. Chapter 13 deals with file input/output operations in VAX DAL, and Chapter 14 describes parameter passing into and out of VAX DAL routines. Chapter 15 explains how to use VAX DAL macros.

Appendix F contains complete listings for the lesson Multiply described in Chapter 5 and for the lesson Icecream used as an example in Chapters 8 and 10. Appendix F also contains the lesson Menu_driver, which uses many of the VAX DAL features described in Chapter 7.

RELATED DOCUMENTS

VAX DAL Reference Manual Order No. AA-K768C-TE	This manual is a complete reference guide to the DIGITAL Authoring Language.
VAX DAL Pocket Reference Guide Order No. AV-DB13B-TE	This manual is a quick reference guide to the commands and features of VAX DAL.
Courseware Authoring System User's Guide Order No. AA-K764C-TE	This manual explains how to use the C.A.S. Delivery System to publish lessons.
Courseware Authoring System System Manager's Guide Order No. AA-K767C-TE	This manual contains information needed for overall administrative control of the C.A.S. Delivery System.

REFERENCES

This manual refers to the following books in the VAX/VMS documentation set:

EDT Editor Manual
Order No. AA-J726A-TC

Introduction to VAX/VMS
Order No. AA-Y500A-TE

VAX/VMS DCL Dictionary
Order No. AA-2800A-TE

VAX/VMS Linker Reference Manual
Order No. AA-Z420A-TE

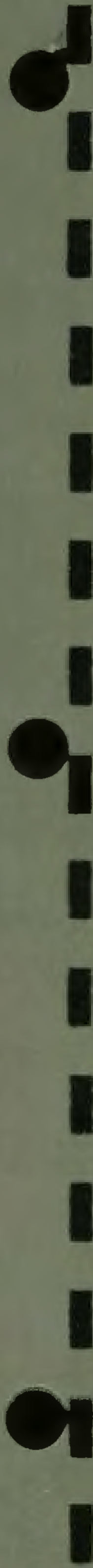
CONVENTIONS USED IN THIS MANUAL

The following printing conventions are used in this manual for the example code.

Uppercase characters	Uppercase characters are used for all language elements that must be entered exactly as shown. These elements are instructions, system variables, system functions, and keywords.
Lowercase characters	Lowercase characters are used for the names of variables and constants defined by the author.
Brackets { }	Brackets enclose optional arguments.
Ellipsis ...	An ellipsis indicates that the previous element can be repeated.
(TAB) (SP) (RET)	These symbols represent three nonprinting characters: horizontal tab generated by the TAB key, space generated by the space bar, and return generated by the RETURN key.

1

Introduction



1 Introduction

A computer-aided-instruction (CAI) lesson is a special kind of computer program. CAI lessons provide students with the opportunity to learn and apply new information and skills. CAI lessons can be written for students at any educational level. A CAI lesson is a type of instructional material. As is true of other types of material — text books, laboratory exercises, games, film strips, and so on — the effectiveness of a CAI lesson depends largely on how well the lesson is planned and how well the plan is implemented.

CAI lessons differ from other instructional materials in the method of presentation and in the methods of structuring material that are possible using a computer. The method of preparing a CAI lesson is also different. Because the lesson is a specialized computer program, it is written as well as executed on the computer. When students take a lesson, they see a series of visual displays on the monitor and type responses to these displays on the keyboard. The displays can take a number of forms. They can explain how to take the lesson or discuss subject matter. In this case, the students' responses generally indicate that they have read the information and are ready to continue. Displays can show a problem to be solved or ask a question to be answered. In this case, the students' responses are answers that are judged and scored. The next display can be some kind of feedback based on the response. Displays can show a game ready for the next move. The students' responses indicate moves, and the next display updates the game.

Planning a lesson includes deciding what each display includes, what student's responses should be, and what should happen next. Right answers can be defined, and an appropriate message can be displayed as feedback. Wrong answers can also be anticipated; for these the feedback can explain the students' mistakes or misconceptions. The question can be repeated, or another question can be displayed.

After the lesson is planned, it is written in VAX DAL.

THE DIGITAL AUTHORING LANGUAGE

Programming a lesson in VAX DAL gives the CAI author complete control over what is in the lesson. The author chooses the instructional format, designs the screen displays that the students see, and controls response judging and feedback.

VAX DAL has the following features:

- Graphics

  The author can design character fonts; draw lines, circles, curves, and dots; and plot graphs. DAL also provides relative screen addresses for graphics, and a capacity to produce full color graphics.

- Flexible Response Judging

  The author defines anticipated right answers and wrong answers, and controls both the exactness with which student responses must match and what happens after a match. The author also controls scoring.

- Structured Lessons

  Each lesson consists of independent units that are executed in the order determined by the lesson. Units can be executed selectively, so that students see only the units they need for learning the material.

Graphics

Lessons written in VAX DAL are executed on a terminal connected to a color or black-and-white monitor that shows the displays. VAX DAL controls placement of text and graphics on the screen.

VAX DAL provides several ways to enhance text displays. As an author, you can use alternate character sets to display mathematical symbols, the Greek or Russian alphabets, or special characters such as game markers. You can modify character size and proportions to display bigger characters, tall, thin characters, or short, fat characters. To emphasize a word, you can use italics or simple graphics such as underlining text or pointing to a word with an arrow. You can divide the screen into separate areas for different purposes and outline the areas. Figure 1–1 shows some of the possibilities.

You can emphasize a word in
several ways:

With ITALICS WITH INVERSE VIDEO

By pointing at it ←

by s t r e t c h i n g it

by <u>underlining</u>

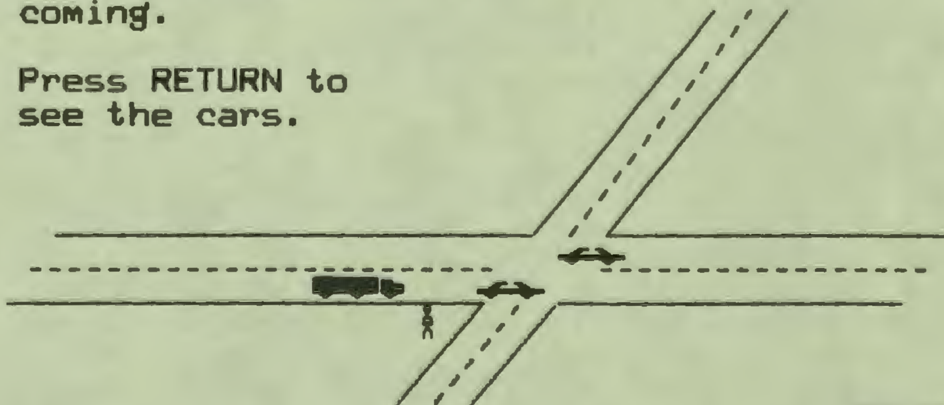
MR-S-2109-82

Figure 1–1
Text Displays

You can draw illustrations made up of lines, dots, circles, boxes, and curves, and draw the same graphic at different locations on the screen. You can shade some areas in your pictures, and both shade and draw lines in different patterns. Figure 1–2 shows some of the possibilities.

It is harder to
cross the street
when cars are
coming.

Press RETURN to
see the cars.



MR-S-2110-82

Figure 1–2
Traffic Lesson

Chapter 3 discusses concepts related to all displays, including screen addresses and color, and explains how to display text. The display of text is illustrated in the sample lesson explained in Chapter 5. Chapter 10 discusses graphics.

Judging the Student's Response

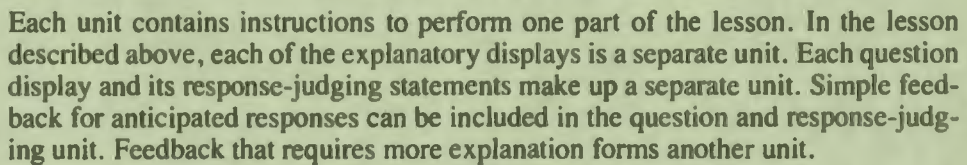
An important part of computer-aided instruction is collecting and evaluating student responses. There are two types of responses: those that answer questions that are part of the educational purpose of the lesson and should be scored, and those that do not. Responses that do not provide answers to questions might either select a subject to be covered next from a menu, ask for instructions to be displayed again, ask for help, or indicate that the student has read the display and wants to continue. As the author, you have complete control over response judging. You can modify how closely the students' responses must match the answers you specify and what happens after an answer is matched. You can end judging after a response judged right, after the first response, or after a specified number of responses that are judged wrong.

Chapter 4 discusses the fundamentals of response judging. Simple response judging and some modifications are illustrated in the sample lesson discussed in Chapter 5. Chapters 8 and 9 discuss more techniques for modifying and controlling response judging.

Structure of a Lesson

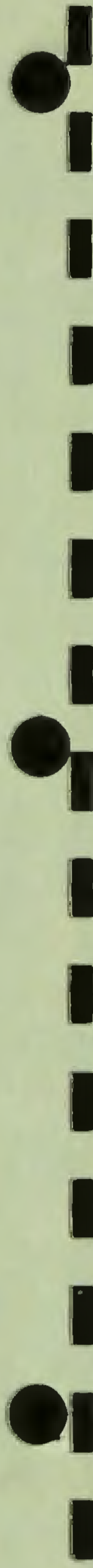
As you plan a lesson, some divisions in the material become apparent. A very short lesson might consist of two displays that explain a concept, followed by three questions that test whether students understand it.

You also plan different feedback for different anticipated responses. Some anticipated responses are specified as right answers. A student who answers all three questions correctly ends the lesson. Other anticipated responses show different degrees of mastery and different kinds of misunderstanding. These anticipated responses are specified as wrong answers. For some, the feedback is simple. Others indicate that the entire concept needs to be explained again in a different way. The lesson written in VAX DAL is also divided. A DAL lesson consists of lesson level and individual units.


Each unit contains instructions to perform one part of the lesson. In the lesson described above, each of the explanatory displays is a separate unit. Each question display and its response-judging statements make up a separate unit. Simple feedback for anticipated responses can be included in the question and response-judging unit. Feedback that requires more explanation forms another unit.

Lesson level contains instructions that control execution of the units.

Chapter 5 discusses one sample lesson in detail, and illustrates how the requirements for the planned lesson are embodied in the structure of the written lesson.



2

Elements of VAX DAL

2

Elements of VAX DAL

This chapter discusses the elements of a lesson written in VAX DAL and defines the terms used for those elements.

LESSON

To a student, a lesson is a series of displays on the screen that show information and questions and require a response. To an author, the same lesson is a series of instructions in VAX DAL that produce the displays, read the student's responses, evaluate them, then produce the next display depending on what the response was. The author writes a lesson. Many students execute the lesson. The difference between the lesson as it is written and the lesson as it executes can be compared to the difference between a recipe and the cooking process.

A recipe, like a lesson, often includes instructions for different circumstances. For example, a recipe can state that either six small eggs or five large eggs are required. The cook decides whether to add small eggs or large eggs depending on the size available. Likewise, in a lesson the author specifies different actions for anticipated responses. One student executing the lesson enters a response, and the lesson determines which specified action to take for this response. Another student executing the lesson gives a different response. This time, the lesson takes a different specified action.

Lessons are divided into lesson level and units. Units are subdivisions of the lesson that can be executed at different times. To continue the cooking analogy, consider a recipe for lemon meringue pie. The pie recipe has four basic instructions: make a single crust, prepare the filling, make the meringue, and bake the pie. The instruction to make the crust refers the cook to a crust recipe on another page. The crust recipe contains detailed instructions for making pie crust.

The main pie recipe can be compared to lesson level, and the crust recipe can be compared to a unit. The crust recipe is written separately and is referred to in the main recipe. At the main recipe's instruction to make crust, the cook refers to the crust recipe, executes the instructions for making crust, and returns to the main recipe for the next instruction.

A VAX DAL lesson as written begins with the lesson-level instructions. Lesson level ends at the beginning of the first unit. Units can be written in any order.

A VAX DAL lesson as executed also begins with the first lesson-level instruction. Execution of the lesson ends after the last lesson-level instruction. The units are executed by DO unit_name instructions at lesson level or in other units. After the instructions in the executed unit are completed, the lesson control returns to the next instruction after DO unit_name.

The instructions that make up a VAX DAL lesson are contained in a computer file, called a source file, that you can read. Before you can execute the lesson you must compile the lesson source file to produce a file that the computer can read. The number of units a VAX DAL source file contains cannot exceed 127. If the source file contains more than 127 units, it must be divided into two or more files that are compiled separately and then linked together. One of these files must contain the lesson-level instructions. The other file(s), which are called modules, contain units.

INSTRUCTION

An instruction in VAX DAL is a word that causes one action. In cooking, mince is one instruction; chop is another instruction; slice is still another. The cook does something different in response to each instruction.

In VAX DAL, WRITE is an instruction that causes some text to be displayed on the monitor screen. AT is an instruction that specifies a location on the screen for the text. Like WRITE and AT, many instructions are English words. The other instructions are mnemonic: that is, words invented to make the action the instructions perform easy to remember. For example, the instruction IF begins a series of instructions that are executed if a condition is true. The ENDIF instruction is mnemonic. It marks the end of the instructions executed if the condition is true.

ARGUMENTS

As you can see from the AT and WRITE examples, the instruction alone does not give complete information. AT requires a location, and WRITE needs some words to display on the screen. The additional information is called an argument. If you want to write "Hello, Susie" at the top center of the screen, you use two instructions with their arguments.

```
AT      220
WRITE   Hello, Susie
```

The instruction AT with the argument 220 tells the lesson to set the location of row 2, column 20 on the screen, and to use this location when it is next instructed to write something. The WRITE instruction with the argument "Hello, Susie" then writes the characters.

VAX DAL is composed of a set of defined instructions. To create a lesson or unit, you select the appropriate instructions from this set. For each instruction the number of arguments is fixed, and each argument's function is precisely defined. Some instructions, such as AT and WRITE, require one argument; others require more than one; and a few do not require arguments. Arguments can be specified in a number of ways.

Keywords

Some instructions require keywords. A keyword is a word that has a special meaning when it is used as an argument. For example, the FCOLOR instruction can use different keywords to select the colors for text and graphics in displays. With the keyword RED as its argument, the FCOLOR instruction causes subsequent text and graphics to use the color red. With the keyword YELLOW, FCOLOR draws text and graphics with the color yellow.

Instructions that require keywords accept only one of the defined keywords.

User-Defined Variables

A variable is a name that identifies information so that you can store the information and retrieve it when it is needed. A variable can be compared to a label on a pigeonhole. You can store information in the pigeonhole, using the variable name to identify its location for later retrieval. When you include a variable as the argument for an instruction, you are telling the lesson to use the information currently in the pigeonhole labeled with the variable name.

Suppose that you would like a lesson to write "Hello, appropriate name" on the screen to greet individual students as they enter the lesson. It is easy to do this by using a variable. You can define your variable as "first_name". The first question of the lesson asks for the student's name. Each student who takes the lesson types a different name. You store whatever the student types in the variable "first_name". For each student the contents of the variable are different. You can now use the variable "first_name" as an argument to a WRITE instruction, and display what the student typed. (Be aware that this technique can produce unexpected results. If an uncooperative student types "What's yours?", the lesson displays "Hello, What's yours?".)

Data Types

You must define each variable you want to use in a lesson. You must also specify the kind of information you are going to store in each variable by identifying its data type. The lesson uses this data type to determine how big the storage area needs to be and how the variable can be used.

The instruction for defining a variable is:

```
DEFINE    variable_name : data_type {,usage_characteristic}
```

The keywords that select one of the possible data types are INTEGER, REAL, STRING, BOOLEAN, and RECORD.

- Integers and Real Numbers

  An integer is a whole number. A real number is a number that contains a decimal point. These two kinds of numbers are stored differently in the computer. When you define a variable as either an integer or a real number, you can use it for arithmetic and for more advanced mathematical functions. You can also display it.

- Strings

  A character is any one of the letters, digits, punctuation marks, and symbols that can be displayed on the screen or that affect the display but are not visible, such as spaces, tabs, and carriage returns. A string is a series of characters.

  Inside the computer, characters are stored as codes. When you press a key on the keyboard, it generates a code. When you use the WRITE instruction to display a character, the monitor receives a code, and draws the corresponding shape on the screen. The digits that allow you to write numbers are coded as characters just as letters and punctuation marks are.

  The size of a string variable can change dynamically during the execution of a lesson. You can use the same string variable to store strings containing different numbers of characters. You can also compare two string variables to see if they are the same.

- Boolean

  Boolean variables have one of two values, true or false. The keywords TRUE and FALSE can be used to assign a value to a Boolean variable.

- Record

  The data type RECORD defines a structure that consists of several component variables. The component variables in a record structure correspond to fields of data in records in an external file. Record structures are used in many file input/output operations. Component variables in record structures can have any of the data types listed above.

You can define structured arrays and tables composed of elements of the integer, real, string, and Boolean data types. Arrays are multidimensional while tables are two-dimensional. Single elements of these data structures are identified by subscripts. Refer to the *VAX DAL Reference Manual* for more information about arrays and tables.

Usage Characteristics

A variable is always defined with a data type. It can also be defined with a special usage characteristic.

The keywords that select one of the usage characteristics are GLOBAL, EXTERNAL, STUDENT, RESTART, PERMANENT, and FUNCTION.

- Global

  Global variables can be referenced by instructions in all of the separately compiled modules of a lesson.

- External

  The external usage characteristic identifies a variable that is also defined as a global variable in a separately compiled part of the lesson. When a lesson is assembled from two or more source files, define the same variable more than once: as a global variable in only one source file, and as an external variable in the other source file(s). This ensures that the variable can be referenced by instructions in the separately compiled parts of a lesson. See the *VAX DAL Reference Manual* for more information about global and external variables.

- Student

  Student variables are those that students can use in their responses. If "pi" is defined as a student variable, a student can type "2*pi*11.2" as a response. Student variables are global variables by default. Do not define student variables with the GLOBAL or EXTERNAL usage characteristic. The *VAX DAL Reference Manual* explains student variables in more detail.

- Restart

  Restart variables are saved in a file if the student stops taking the lesson before the lesson is finished. When the student finishes the lesson, the restart variable file contains a record of the student's performance. Restart variables are global variables by default. Do not define restart variables with the GLOBAL or EXTERNAL usage characteristic. The *VAX DAL Reference Manual* explains restart variables in more detail.

- Permanent

  Like restart variables, permanent variables are saved when the student ends the lesson. Each lesson has one set of permanent variables that is referenced during consecutive executions of the lesson. Permanent variables are global variables by default. Do not define permanent variables with the the GLOBAL or EXTERNAL usage characteristic. The *VAX DAL Reference Manual* explains permanent variables in more detail.

- Function

  The usage characteristic FUNCTION defines the variable as the name of a user-defined function. User-defined functions are explained in the *VAX DAL Reference Manual*.

System Variables

DAL maintains system variables that a lesson can refer to. The values of system variables are modified by instructions in the lesson. System variables fall into four different categories.

- Scoring System Variables

  These variables contain information related to student scores. Scoring system variables contain information such as the number of correct responses a student makes, and the number of correct responses a student makes on the first try.

- Response-Related System Variables

  These variables contain information about actual student responses, such as a student's response time or the number of characters in a response. Response-related variables also place restrictions on how a question can be answered; restrictions such as a time limit for answering a question, or the maximum number of characters allowed in a response.

- Graphics System Variables

  These variables keep track of graphics information, such as the current cursor location, the current text size, and the current writing color.

- Miscellaneous System Variables

  These variables store generally useful information, such as the amount of time that has elapsed since the student began the lesson.

Specific system variables are discussed throughout this manual. Appendix C contains a list of all system variables.

Constants

Constants are elements whose values do not change during lesson execution.

The AT and WRITE instructions shown above use constants as arguments to write "Hello, Susie" at row 2, column 20. To select a different location, you must use a different row number and/or a different column number. To write a different message, you must change the argument to WRITE.

Constants can have the data types integer, real, string, or Boolean. A number without a decimal point, such as 100 or –15, is an integer. A number with a decimal point and a decimal part, such as 0.5 or 25.93, is a real number. Most string constants must be enclosed in double quotation marks (" "). A string constant used as an argument to a WRITE or SPEAK instruction is the exception to this rule.

You can also define a named constant just as you can define a variable. For example, in a lesson on calculating the circumference and area of circles, the real number 3.14159 is used in many calculations. You can enter this constant every time it is required; but defining the named constant pi is easier. Then whenever you write "pi" as an argument, the lesson uses the specified number.

Named constants are defined as follows:

```
DEFINE    name=value
```

The data type of the value determines the data type of the named constant. The constant pi in the following example is a real number.

```
DEFINE    pi = 3.14159
```

Functions

A function is a data manipulation that is built in to DAL. For instance, there are mathematical functions for finding sines and cosines and string functions for finding a character or a word in a student's response.

Functions consist of the function name and arguments. As with instructions, the number and type of arguments depend on the function. The function INT converts a real number to an integer. The argument, then, must be a real number. The system variable SCORE contains the student's score as a real number. The function INT(SCORE) returns the score as an integer.

Appendix B lists all system functions and their arguments.

Expressions

Expressions are combinations of constants, variables, functions, and operators. The operators define the relationship of the other elements. When the lesson is executed, expressions are evaluated.

Numeric operators perform arithmetic. The four numeric operators are:

+ addition
– subtraction
* multiplication
/ division

A simple expression is x + y. When this expression is evaluated, the current value of the variable x is added to the current value of the variable y. The expression can be used as an argument. The instruction AT x + y first evaluates the expression, then selects the address defined by the result of the evaluation.

In addition to numeric operators, there are also relational operators. The relational operator <> means not equal to. The expression SCORE <> 100 is a Boolean expression. SCORE is a system variable that contains the student's current score. The expression is true when the score is any number except 100, and false when the score is 100.

The relational operator = means equal to. The expression RESPONSE = "Washington" is a Boolean expression that is true when the string in the system variable RESPONSE is equal to the string "Washington" and false when it is not. (The system variable RESPONSE contains the student's most recent response.) The only relational operators that can be used with a string are equal to (=) and not equal to (<>).

The other relational operators that can be used with integer, real, and Boolean variables are listed below.

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Boolean expressions are often used to define a condition to be tested.

Parentheses are used in expressions with the usual algebraic function of defining a quantity.

Appendix D lists all operators and their order of precedence. Expressions are explained throughout this manual when they appear in the examples.

Default

A default is a value used by the lesson unless one of the instructions in the lesson overrides it. For example, the default color for the background color on a color monitor is dark. When you use the instruction BCOLOR, you can override the default to select a different background color.

SELECTING NAMES

The lesson and any lesson units, modules, variables, or constants require unique names that you select. The choice of names is restricted by the following rules. If you do not observe these rules, you cannot compile the lesson (see Chapter 6).

- Lesson name, module names, and unit names

  The lesson name and all unit, condition unit, and module names can contain only alphanumeric characters and can be no more than 31 characters long. Names cannot begin with a dollar sign ($) or with the prefix DAL_. The lesson name, unit names, condition unit names, and module names must be unique. They cannot be the same as the names of system variables, user-defined variables, or system functions.

- Variable names and constant names

  Variable names cannot be longer than 32 characters. All variable names must begin with a letter and can contain letters, digits, and the punctuation mark underscore (_). The name of each variable used in the lesson must be unique. Variables defined at lesson level can be used at lesson level and in different units.

  The rules for constant names are the same as the rules for variable names.

It is strongly recommended that the lesson name be the same as the name of the source file containing the lesson instructions.

Although the rules allow you to name variables with the same names as system variables and system functions, do not use these names. When the lesson is executed, the system variables and functions always take precedence. If you use the same names, your variables are not used.

The rules allow you to define variables with the same name at lesson level and in different units. Variables defined at lesson level are available throughout the lesson. Variables defined in one unit are available only in that unit. When a unit executes, DAL first references the variables defined in the executing unit. If the variable is not defined in the unit, the lesson references a lesson-level variable.

Any variables that are used both in the main lesson module and in separately compiled modules should be defined more than once. In one module, define the variables with the usage characteristic GLOBAL; in the other modules, redefine the same variables with the usage characteristic EXTERNAL.

Generally, it is better to use different names for all variables. Then, as you are writing the lesson, you always know whether you are using unit-level variables or lesson-level variables.

Choose variable names that are easy to remember and have a clear relationship to the information stored in the variable. Relatively long variable names using the underscore character are usually clearer than very short variable names. The variable name egg_size is clearer than the variable name egsz.

Because the dollar sign is used for many operating system names, it should not be used in variable names.

SYNTAX

Syntax is the required order and punctuation of DAL elements in a line of code.

Generally, a line of code begins with an instruction. The instruction is followed by one tab or space character and by the arguments required for that instruction. The arguments are separated by required punctuation marks. Spaces and tabs can be used between arguments to make the code more readable. Do not begin a line of code with a tab or space character.

You can enter comments in your code by beginning a line with either a semicolon or an exclamation point, or by using two dollar signs after the arguments to an instruction. The semicolon, exclamation point, or dollar signs cause comments to be ignored when the lesson is compiled. Since the code itself is abbreviated, comments are a way of making the instructions easier to understand.

You can write instructions, variable names, function names, and keywords in either uppercase or lowercase characters. Case is ignored. The instruction WRITE is the same as the instructions Write or write. The variable name X_coordinate is the same as the variable name x_coordinate.

Case is preserved in string constants and in text displayed with the WRITE instruction.

The example below shows two lines of code. The symbol (TAB) represents the nonprinting character generated by the TAB key. The symbol (SP) represents the nonprinting character generated by the space bar. The symbol (RET) represents the nonprinting character generated by the RETURN key.

```
FCOLOR(TAB)MAGENTA(RET)
LINE(TAB)300,100(SP);(SP)500,100(RET)
```

Each line begins with an instruction. Either the (TAB) as shown or a (SP) is required to separate an instruction and its arguments. The argument MAGENTA is a keyword that selects magenta as the foreground (writing) color. The (RET) at the end of the line is required. In the second line, the two constants 300,100 are one argument defining the starting address of a line. The comma is a required syntax element. The (SP) is optional, and makes the line easier to read. The second argument is 500,100 and defines the ending address of the line. The semicolon between the arguments is a required syntax element.

Dot Indentation

Dot indentation is a special syntax element used with some instructions to show the structure of a series of instructions. When dot indentation is required, the first character in a line is a period; and the second is a tab character.

For example, the LOOP instruction begins a series of instructions that is repeated as long as the Boolean expression that is the argument to LOOP is true. The instruction ENDLOOP marks the end of the instructions to be repeated. The instructions in between use dot indentation to show that they are inside the loop.

USING THE ELEMENTS OF VAX DAL

This section shows some lines in DAL and explains the elements and syntax.

As the instructions are discussed in the rest of the manual, the arguments they require and their syntax is also explained. This information is also summarized in the appendixes for easy reference. Appendix A lists all DAL instructions, including those not discussed in this manual. Appendix B lists all system functions. Appendix C lists system variables. Appendix D lists operators and operator precedence for expression evaluation. Appendix E lists syntax symbols.

The following unit in DAL is a review unit that displays the score for one set of arithmetic problems.

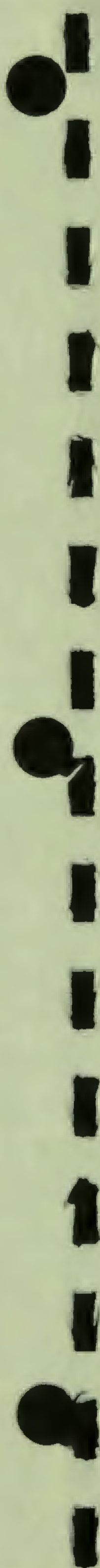
```
; Comment. The semicolons define these two
; lines as a comment.
! Exclamation points also define lines as comments.
! Comments are useful for authors, but have no effect
! when the lesson is executed.

UNIT     review   $$ The double dollar sign indicates that
                  $$ what follows is also a comment. The instruction
                  $$ UNIT begins a unit whose name is review.
ERASE             $$ The ERASE instruction erases the screen.
IF       NNO = 0
.        AT       1020    $$ 1020 is a constant, defines place to write.
.        WRITE    Very good;
                  You got all the problems right.
ELSE
.        AT       1020
.        WRITE    Your score is <<S,INT(SCORE)>>.
ENDIF

PAUSE             $$ PAUSE is an instruction to pause in the lesson
                  $$ until the student presses the return key.
```

The instructions IF, ELSE, and ENDIF are part of a structure that requires dot indentation as shown. The argument to IF is a Boolean expression consisting of the system variable NNO, the operator = , and the constant 0. NNO contains the number of wrong answers the student has given. When this expression is evaluated, it is true if the number of wrong answers is equal to 0. When the expression used as the argument to IF is true, the instructions between IF and ELSE are executed.

These instructions write the text that is the argument to the WRITE instruction. There are two tab characters before the second line of text to be displayed. This is the proper syntax for writing a block of text.

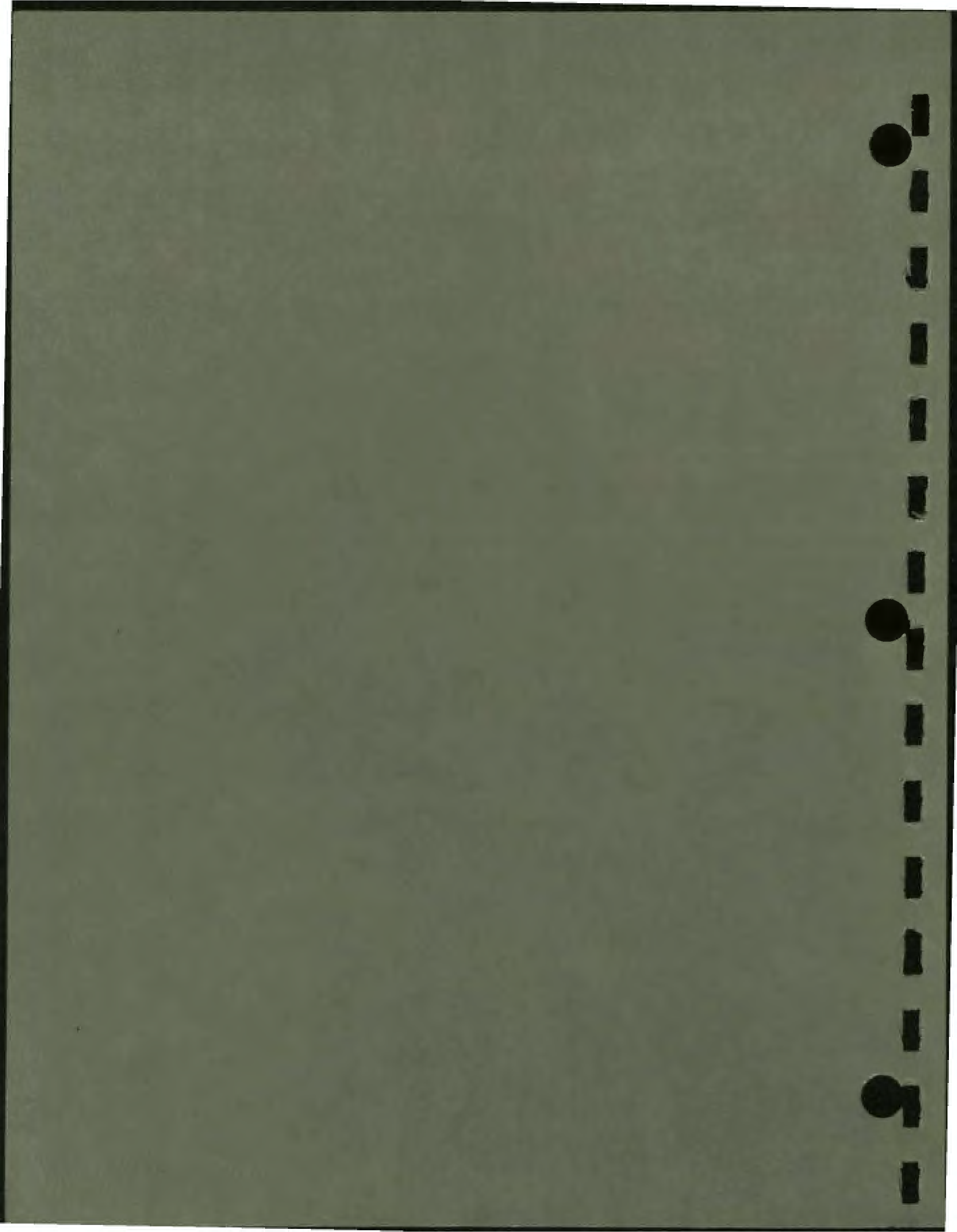
The instructions following ELSE are executed when the argument to IF is false: that is, when the student gave at least one wrong answer. The argument to WRITE uses the system variable SCORE (SCORE contains the student's total score). This is a real number, so the function INT is used to convert it to an integer so that a decimal point and zeros to the right of the decimal point are not displayed. INT(SCORE) is the syntax for a function. The S and the double angle brackets enclosing the function are the required syntax for specifying that the value of a variable should be displayed.

The instruction ENDIF is the end of the sequence begun by IF. The dot indentation shows this structure.



3

Writing on the Screen



3

Writing on the Screen

All lessons display text on the screen, and many lessons display some form of graphics. This chapter explains the following basic concepts that apply to all screen displays:

- Screen addresses
- Current display attributes
- Displaying text on the screen
- Text sizes
- Color
- Modes

Chapter 10 explains the graphics instructions used to draw pictures and the instructions for displaying italics and rotated text.

SCREEN ADDRESSES

Displays on the monitor screen consist of illuminated dots. Line graphics are drawn by illuminating the dots for a circle, a box, or a line.

Characters are displayed as dots in a character cell. A character cell is like a child's alphabet block. It is rectangular and consists of dots for the character — that is, for A, a, B, b, and so on — and dots for the background. The entire character cell is displayed. Either the dots for the character or the dots for the background can be illuminated.

All instructions that display text or graphics on the screen require screen addresses. To display a line, the addresses of the beginning of the line and the end of the line are required. To draw a circle, the address of its center and the number of dots for the radius are required. To display text, the address of the top left corner of the character cell containing the first character in the string is required.

Screen addresses are specified in one of three coordinate systems. Row-and-column coordinates divide the screen into units the right size for a character cell. Fine coordinates divide the screen into units the size of a displayable dot. Normalized coordinates specify addresses as a proportional distance on the screen.

Row-and-Column Coordinates

Figure 3–1 shows the screen and the number of rows and columns. The screen is divided vertically into 24 rows and horizontally into 80 columns. Rows are numbered from 0 to 23 and columns from 0 to 79. Each position on the grid is specified by a row number and a column number. Each position is the right size for one size 1 character cell. The text in Figure 3–1 is size 1, which is the default character size.

Most of the text in Figure 3–1 is displayed in normal mode. In normal mode, the dots that make up the character are illuminated in the foreground color. The dots for the rest of the cell are illuminated in the background color and, therefore, are not visible. Figure 3–1 displays two characters in inverse mode so that the character cell is visible. In inverse mode, the dots for the character are illuminated in the background color and the dots for the rest of the character cell are in the foreground color.

MR-S-2111-82

Figure 3–1
Row-and-Column Addresses

Text is displayed with the AT and WRITE instructions. The argument to the AT instruction selects the screen address. The argument to the WRITE instruction specifies the characters.

Row-and-column addresses are selected with a three- or four-digit number. The rightmost two digits select a column. For columns below 10, a leading zero must be used. The left digit or digits select a row, and a leading zero is not necessary.

The following instructions display the text in Figure 3–1.

```
AT      709
WRITE   This line begins at
        row 7 column 9.
        The address is 709.
AT      1540
WRITE   row 15
LINE    1640;1647
AT      2119
WRITE   column 19
BOX     2119;2228
```

```
AT      550
WRITE   These two character cells are
        at 850 and 860.
MODE    INVERSE
AT      850
WRITE   a
AT      860
WRITE   b
MODE    NORMAL
PAUSE
```

Row-and-column coordinates are usually used with text. Row-and-column coordinates can be used with graphics, and are especially useful when the graphics are closely related to the text.

With the graphics instructions, row-and-column addresses select the top left corner of the character cell. The two addresses used with the BOX instruction above select opposite corners of the box. As you can see from Figure 3–1, one corner of the box is located at the top left corner of the character cell at row 21, column 19; the opposite corner is located at the top left corner of the character cell at row 22, column 28.

The BOX instruction also illustrates the syntax for graphics instructions. The two arguments to the BOX instruction specify opposite corners of the box – in this case the upper left corner and the lower right corner. Because row-and-column addresses specify the positions on the screen as one integer, there is no punctuation in a row-and-column address. The semicolon separates the two addresses.

Fine Coordinates

Fine coordinates divide the screen into smaller units than row-and-column coordinates. In fine coordinates, there are 480 vertical units, numbered from 0 to 479. Depending on the terminal model in use, there are either 767 horizontal units numbered from 0 to 766, or 799 horizontal units numbered from 0 to 798. Table 3–1 lists the number of horizontal fine address units that each DIGITAL terminal model supports.

Table 3–1: Horizontal Fine Address Units

Terminal model	Number of units
GIGI (VK100) VT125 DECmate II DECmate III Professional	767
VT240 VT241 Rainbow	799

Each horizontal address specifies one displayable dot. However, there are two addresses for each displayable dot in the vertical direction. Each odd-numbered address and the even-numbered address below it select the same dot.

The addresses in the horizontal and vertical directions divide the screen into units of the same size. A line drawn horizontally from x-coordinates 0 to 100 is the same length as a line drawn vertically from y-coordinates 0 to 100. The actual length of such a line depends on the size of the monitor.

Addresses in fine coordinates are given as two integers separated by a comma. The first integer specifies the horizontal position (x-coordinate). The second integer specifies the vertical position (y-coordinate).

Figure 3–2 shows the fine coordinate system.

0 x-direction 767

The screen addresses of the two points
of the horizontal arrow are
25,20 and 740,20.

y-direction

The screen addresses of the two points of the vertical
arrow are 25,35 and 25,450.

The center of the circle below is at 525,425.
Its radius is 60.

479

MR-S-2112-82

Figure 3–2
Fine Screen Addresses

The following instructions draw Figure 3–2 except for the grid. The grid lines are 50 units apart.

```
ERASE
WRITE   0                                    x-direction
AT      082
WRITE   767
AT      500
WRITE   y
        -
        d
        i
        r
        e
        c
        t
        i
        o
        n
```

```
AT      2300
WRITE   479
VECTOR  350,20;25,20:0.05
VECTOR  350,20;695,20:0.05
AT      430
WRITE   The screen addresses of the two points
           of the horizontal arrow are
              25,20 and 740,20.

VECTOR  25,250;25,35:0.07
VECTOR  25,250;25,450:0.07
AT      1010
WRITE   The screen addresses of the two points of the vertical
        arrow are 25,35 and 25,450.

AT      1530
WRITE   The center of the circle below is at 525,425.
        Its radius is 50.
DOT     525,425
CIRCLE  525,425:50
```

In this example, row-and-column addresses are used with text, and fine addresses are used with the graphics instructions. The graphics instructions show the syntax for specifying addresses in fine coordinates.

The DOT instruction illuminates one dot and requires one address as its argument. The x-coordinate and y-coordinate defining the address are separated by a comma.

The CIRCLE instruction also requires one address — the center of the circle — as the first argument. The x-coordinate and the y-coordinate are again separated by a comma. The second argument is the radius, and it is given in screen dots. The colon between the two arguments is the standard syntax for separating addresses from other arguments to graphics instructions.

The VECTOR instruction draws an arrow. The VECTOR instruction requires three arguments: the address of the tail of the arrow, the address of the point, and the size of the arrow head. Each address consists of an x-coordinate and a y-coordinate separated by a comma. A semicolon separates the two addresses. As with the CIRCLE instruction, the colon separates the address arguments and other arguments. The last argument to the VECTOR instruction specifies the size of the arrow head as a proportion of the length of the arrow.

Normalized Coordinates

With the normalized coordinate system, screen addresses are also specified in x- and y-coordinates. Each coordinate, however, is a real number between 0 and 1.0. Normalized coordinates specify locations as a proportion of the total horizontal distance across the screen or the total vertical distance down the screen.

Normalized coordinates are written as real numbers. The lowest number is 0.0. The address 0.0,0.0 is at the top left corner of the screen.

Normalized fine coordinates are proportional: that is, 0.25 is one-fourth of the distance across the screen or down the screen, and 0.75 is three-fourths of the distance.

The instruction LINE 0.0,0.0;0.5,0.0 draws a horizontal line from the top left corner of the screen to the dot at the top of the screen and half way across. The instruction LINE 0.0,0.0;0.0,0.5 draws a vertical line from the top left corner to the dot at the left of the screen and half way down. Because the screen is not square, these two lines are not the same length.

The instruction BOX 0,0;0.999,0.999 draws a box around the outside edge of the displaying area on the screen. In normalized coordinates, the first two digits to the right of the decimal point always make a visible difference in the location on the screen. The x-coordinates 0.15 and 0.16 specify different locations. The third digit to the right can make a visible difference. The three y-coordinates 0.753, 0.754, and 0.755 specify the same location on the screen. The y-coordinate 0.756 specifies a different location.

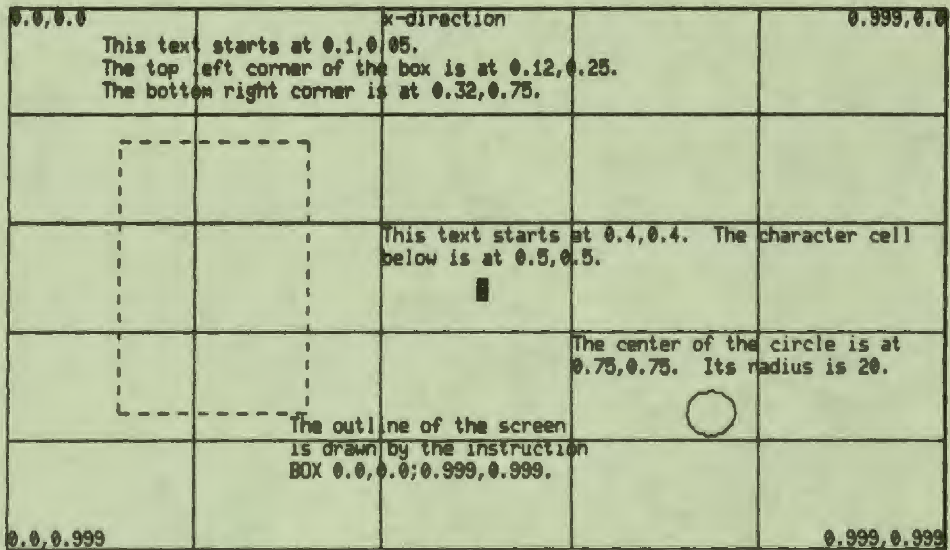


Figure 3–3
Normalized Screen Coordinates

The following instructions draw Figure 3–3 except for the grid. In this figure the grid in both directions shows the locations of the 0.0, 0.2, 0.4, 0.6, and 0.8 normalized addresses.

```
ERASE
WRITE   0.0,0.0                                      x-direction
AT      076
WRITE   0.999,0.0
AT      2300
WRITE   0.0,0.999
AT      2374
WRITE   0.999,0.999
AT      0.4,0.4
WRITE   This text starts at 0.4,0.4. The character cell
        below is at 0.5,0.5.
MODE    INVERSE
AT      0.5,0.5
WRITE   c
MODE    NORMAL
```

```
AT       0.1,0.05
WRITE    This text starts at 0.1,0.05.
         The top left corner of the box is at 0.12,0.25.
         The bottom right corner is at 0.32,0.75.

PATTERN  DASH
BOX      0.12,0.25;0.32,0.75
PATTERN  SOLID
AT       0.6,0.6
WRITE    The center of the circle is at
         0.75,0.75. Its radius is 20.
CIRCLE   0.75,0.75:20
AT       0.3,0.75
WRITE    The outline of the screen
         is drawn by the instruction
         BOX 0.0,0.0;0.999,0.999.
BOX      0.0,0.0;0.999,0.999
```

Screen Address Summary

The syntax of the arguments for screen addresses determines the coordinate system. You are responsible for specifying addresses that are on the screen. DAL displays text and graphics at the location you specify. There is no check to determine if this location is visible on the screen.

You can use any of the three coordinate systems with any of the graphics instructions and with the AT instruction to select a location for writing text. The form of the arguments to these instructionsdetermines the coordinate system.

In row-and-column coordinates, each address required by the instruction is a three- or four-digit integer. The two rightmost digits select the column, and the left digit or digits select the row. Row-and-column coordinates used with graphics instructions select the dot at the top left corner of the character cell.

In fine coordinates, each address is specified by two integers separated by a comma. The first integer is the x-coordinate. The second integer is the y-coordinate. The instruction AT 100,50 selects the position 100 dots to the right and 50 dots down from the top left corner of the screen. When an instruction requires two or more addresses, the x- and y-coordinates for each address are separated by a semicolon. When fine addresses are used for displaying text, the top left corner of the character cell is at the specified address.

To convert from row-and-column addresses to fine addresses, multiply the row number by 20 to determine the y-coordinate. Multiply the column number by 9 to determine the x-coordinate. For example, the row-and-column address 1020 converts to the fine address 180,200. Column 20 * 9 equals the x-coordinate. Row 10 * 20 equals the y-coordinate.

In normalized fine coordinates, each address is specified by two real numbers (numbers with a decimal point) between 0.0 and 1.0. The first number is the x-coordinate. The second number is the y-coordinate. Normalized coordinates specify a proportion of the total distance across the screen and down the screen.

The x-coordinate and the y-coordinate that select one address must be both integers or both real numbers. You cannot mix coordinate systems in one address. When an instruction requires more than one address, however, each address can use a different coordinate system.

The following instructions are acceptable:

```
BOX     0,0 ; 0.999,0.999       $$ Fine coordinates (integers) for
                                $$ upper left corner; normalized
                                $$ coordinates (real numbers)
                                $$ for lower right.

LINE    300 ; 0,400             $$ Row and column coordinates (one
                                $$ integer) for beginning of line;
                                $$ fine (two integers) for end.
```

The following instruction is not acceptable.

```
AT      100,0.5                 $$ x-coordinate is integer; y-coordinate
                                $$ is real number. Mixes fine and
                                $$ normalized coordinates in one address.
```

CURRENT ATTRIBUTES

The concept of current values of attributes is important for writing text and graphics on the screen. There are a number of attributes for all displays. The screen itself is a color — the current background color. The text or graphics is a color — the current foreground color. Text is a size — the current text size. Displaying anything on the screen changes the current location. When a lesson begins, these attributes have a default current value. Instructions in the lesson change the values. Subsequent displays then use the new current values.

When a WRITE instruction is executed, the text begins at the current location. The characters are displayed in the current text size, foreground color, and display mode.

Figure 3–4 shows some text in different sizes and display modes.

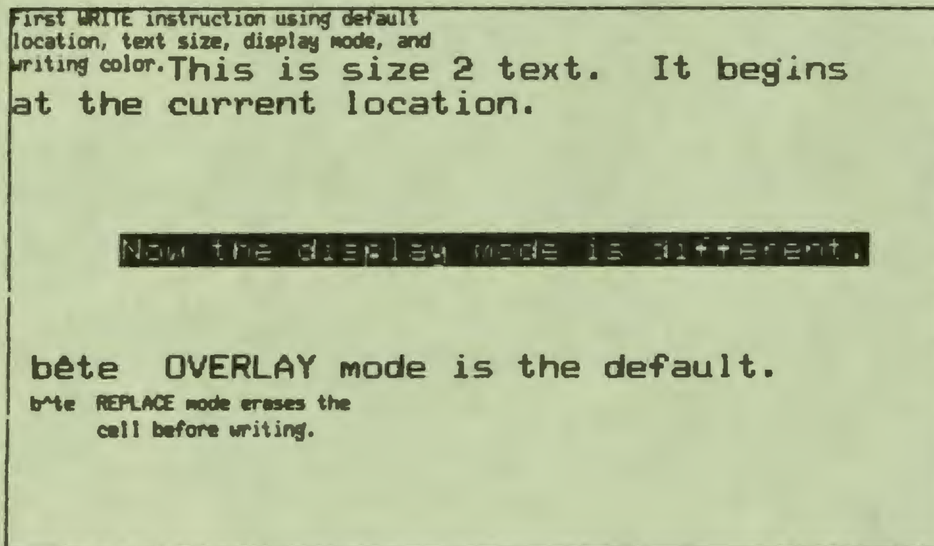


Figure 3–4
Current Location, Size, and Mode

The following instructions draw Figure 3–4.

WRITE	First WRITE instruction using default location, text size, display mode, and writing color.
SIZE	2
WRITE	This is size 2 text. It begins at the current location.
AT	1010
MODE	INVERSE
FCOLOR	YELLOW
WRITE	Now the display mode is different.
MODE	NORMAL

```
AT      1502
WRITE   be
AT      1504
WRITE   the OVERLAY mode is the default.

AT      1702
FCOLOR  WHITE
SIZE    1
MODE    REPLACE
WRITE   be
AT      1703
WRITE   the REPLACE mode erases the
        cell before writing.
BOX     0,0;0.999,0.999
```

Consider the idea of the current text size, and look at the order of instructions. When a lesson begins, size 1 is the current text size by default. Because there is no SIZE instruction before the first WRITE instruction, the first text is written in size 1 characters. The instruction SIZE 2 changes the current text size to size 2. Because size 2 is now the current text size, the WRITE instructions following the SIZE 2 instruction write text in size 2 characters. The SIZE 1 instruction later in the lesson again changes the current text size to size 1. Because the current size is now size 1, the WRITE instructions following the SIZE 1 instruction write text in size 1 characters.

The same idea of a current value applies to the location, the display mode, the writing color, and other attributes of screen displays.

Current Location

The current state of most graphics attributes changes only when the appropriate instruction is executed. The SIZE instruction changes the current text size; the FCOLOR instruction changes the current foreground color, and so on. The current location is the exception. The current location changes whenever a WRITE instruction or any graphics instruction displays text or graphics on the screen.

When a lesson begins, the current location is the top left corner of the screen. The AT instruction sets the current location to the address specified. The text displayed by the next WRITE instruction begins at the current location.

The instructions that draw Figure 3–4 show how the current location works with text. Because there is no AT instruction before the first WRITE instruction, the text specified by the first WRITE instruction begins at the default current location, which is the top left corner of the screen. As each character is written, the current location changes. After each character, the current location is the top left corner of the next character cell.

Between the first and second WRITE instructions, the current text size changes, but the current location does not. There is no AT instruction. So the text displayed by the second WRITE instruction begins at the current location, which is the top left corner of the next character cell to the right of the last character on the screen.

The instructions that draw Figures 3–2 and 3–3 show another instruction that changes the current location. Both sets of instructions begin with an ERASE instruction and a WRITE instruction. There is no AT instruction in between. The ERASE instruction with no arguments erases the entire screen and sets the current location to 0,0, so the next WRITE instruction begins writing at the top left corner.

All graphics instructions also change the current location. With LINE, BOX, CURVE, and VECTOR, the current location after the figure is drawn is the last point displayed. With CIRCLE, the current location is the center of the circle. If you display text after a graphics instruction, the first text cell begins at the current location.

DISPLAYING BLOCKS OF TEXT

In the examples above, there are several WRITE instructions that are followed by more than one line of text. These lines actually contain the characters (TAB) and (RET). If you could see these characters, the instructions to write a block of text would look like this:

```
AT(TAB)510(RET)
WRITE(TAB)First line(RET)
(TAB)Second line(RET)
```

After you have specified one line of text, you can continue the same block by using the (TAB) to line up the second and subsequent lines.

The AT instruction sets a left margin for a text block as well as specifying the current location. In Figure 3–4, the first text block begins at location 000 (row 0 and column 00). Because there is no AT between the first two WRITE instructions, the first character of the second text block is displayed at the current location, which is the top left corner of the next character cell to the right of the last character on the screen. The second line of the second block, however, begins at column 00. Since the left margin has not been reset, the second line of text begins at the current left margin.

To display a blank line in a text block, the line in the source file must have at least one character. Use a TAB or a space. Completely blank lines are ignored when the lesson is compiled.

The syntax for a WRITE instruction inside a structure that requires dot indentation is shown below. The period is used only on the line with WRITE. Lines of code that specify text for the rest of the block require the TAB character for spacing, but cannot have a period. If the WRITE instruction follows dot indentation (as it does in the example shown below), indent the continuation lines such that the continued text starts directly below the first line of text.

The following example shows two levels of dot indentation.

```
LOOP    TEST < 10
.       DO      QUESTION
.       IF      SATISFIED = 1
.       .       WRITE   Very good.
.       ELSE
.       .       WRITE   You missed that one.
                        Take a little more time
                        for the next one.
.       ENDIF
ENDLOOP
```

DISPLAYING VARIABLES

The WRITE instruction can also display the value of a variable or an expression. The syntax is:

```
WRITE   <<S,variable_name>>
WRITE   <<T,variable_name,TD,RD>>
```

The double angle brackets indicate that the current value of the variable is to be displayed. The character following the angle brackets is a format selector. There are two formats: string format selected by the S, and tabular format selected by the T.

Figure 3–5 shows variables displayed in both formats.

```
STRING FORMAT                        TABULAR FORMAT

onetwo                                1000   25
one, two, and three                     1000     25

                                     The number 1000 is too big to
                                     fit in 3 character positions.
one                                  ***
two
three                                Tabular format truncates digits
                                     after the decimal point or adds
                                     zeros to meet the format.
The answer is one.                    1.52      1.5
                                     70.50     70.5
integers in string format
100025
1000 25


real numbers in
string format
1.525000 70.50000
```

MR-S-2117-82

Figure 3–5
Displaying Variables

The following instructions draw Figure 3–5. The first five WRITE instructions show string format with string variables, integer variables, and real variables. The last three WRITE instructions show tabular format with integer and real variables.

```
LESSON   writpic
ERASE
DEFINE   string1,string2,string3:string
ASSIGN   string1 := "one"
ASSIGN   string2 := "two"
ASSIGN   string3 := "three"
AT       101
WRITE    STRING FORMAT
AT       301
WRITE    <<S,string1>><<S,string2>>
         <<S,string1>>, <<S,string2>>, and <<S,string3>>
```

```
AT      801
WRITE   <<S,string1>>
        <<S,string2>>
        <<S,string3>>
AT      1301
WRITE   The answer is <<S,string1>>.

DEFINE  int1,int2:integer
ASSIGN  int1 := 1000
ASSIGN  int2 := 25

AT      1501
WRITE   integers in string format
        <<s,int1>><<s,int2>>
        <<s,int1>> <<s,int2>>
DEFINE  real1,real2:real
ASSIGN  real1 := 1.525
ASSIGN  real2 := 70.5

AT      2001
WRITE   real numbers in
        string format
        <<s,real1>> <<s,real2>>

AT      140
WRITE   TABULAR FORMAT
AT      340
WRITE   <<T,int1,5,0>><<T,int2,5,0>>
        <<T,int1,7,0>><<T,int2,7,0>>

AT      640
WRITE   The number 1000 is too big to
        fit in 3 character positions.
        <<T,int1,3,0>>

AT      1040
WRITE   Tabular format truncates digits
        after the decimal point or adds
        zeros to meet the format.
        <<T,real1,5,2>><<T,real1,10,1>>
        <<T,real2,5,2>><<T,real2,10,1>>
BOX     0,0;0.999,0.999
PAUSE
ENDLESSON
```

Variables can be displayed as part of a text string or a text block. In string format, no spaces are inserted before or after the variable. The first WRITE instruction in the example displays two lines of text. In the first line, there are no spaces between the angle brackets for the two variables. In the second line, there are commas, spaces, and a word between the variables. Figure 3–5 shows the difference in the two displayed lines.

String format can also be used to display integer and real variables. No spaces are inserted between variables.

Tabular format specifies spacing for both integer and real variables. For real variables, tabular format also truncates or adds zeros to the right of the decimal point.

Tabular format requires two arguments after the variable name. The first argument specifies the total number of columns reserved for the number, including a column for a minus sign. For real numbers, the total number of columns also includes a column for the decimal point. The total number of columns begins at the current location. The number is right justified, and columns on the left are filled with spaces.

The second argument specifies the number of digits to the right of the decimal point. For integers, this argument is 0; by definition, integers are whole numbers and have no fractional part. Real numbers either are truncated or have zeros added to display the specified number of digits.

If the number is too big to be displayed in the specified number of digit positions, the lesson displays an asterisk in each digit position.

TEXT SIZE

Figure 3–4 shows text in size 1 and size 2. Size 2 characters have the same proportions as size 1 characters. The instruction SIZE 2 has one integer as its argument, and changes both the height and width of the characters.

Size 1 is the default character size. Whenever the SIZE instruction is executed with only one argument, resulting text characters have the same proportions as size 1 characters.

You can also specify two numbers to change the height and width independently. This lets you display tall, narrow characters or short, wide ones. Figure 3–6 shows some possible text sizes.

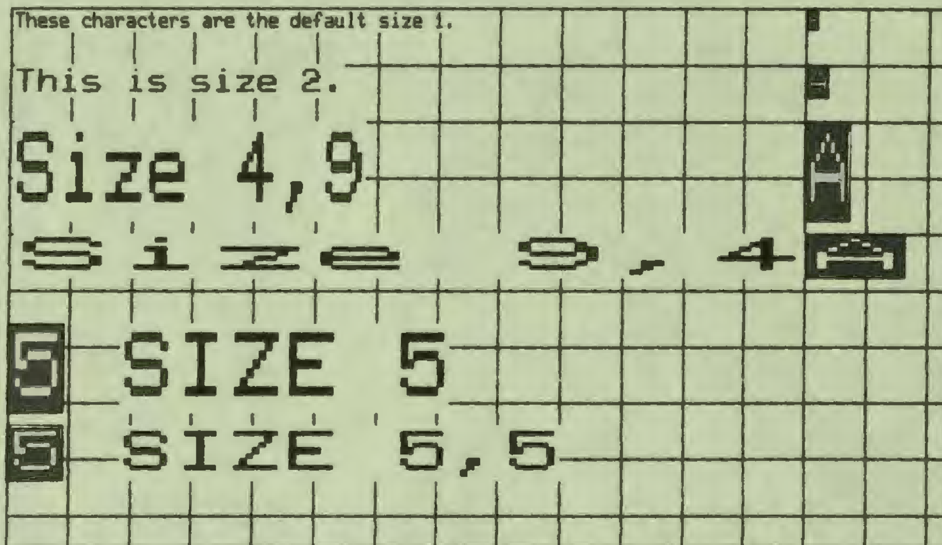


Figure 3–6
Text Sizes

The two formats for the SIZE instruction are:

SIZE	size
SIZE	x-size,y-size

If the SIZE instruction is executed with only one value as its argument, the value specifies the x-size (width) of subsequent text characters. The height of the characters is calculated from the default y-size associated with that x-size. Default y-sizes associated with x-sizes 1 through 16 are listed in the table below.

Table 3–2: Default Character Heights

X-Size	Y-Size (default)	X-Size	Y-Size (default)
0	1	9	14
1	2	10	15
2	3	11	17
3	5	12	18
4	6	13	20
5	8	14	21
6	9	15	23
7	11	16	24
8	12		

If the SIZE instruction is executed with two arguments, the first argument specifies x-size, and the second specifies y-size for subsequent characters.

The two ways of specifying size do not result in characters that are the same dimension. This is true even when the same integer is used for size and for both x-size and y-size. The width of the characters is the same, but the character heights are different. Figure 3–6 shows characters specified as SIZE 5 and SIZE 5,5. The size 5 characters have the same height:width proportions as the default size 1 characters. The size 5,5 characters are shorter in proportion to the width.

Text character height and width are calculated in the same units as fine addresses. Width is the value specified as the x-size multiplied by 9 units. Height is the value specified as the y-size multiplied by 10 units. Using these formulas, the instruction SIZE 3,4 produces text characters 27 (3 * 9) units wide and 40 (4 * 10) units high. The instruction SIZE 4 produces characters 36 units wide and 60 units high (because the default y-size associated with size 4 is 6). Character dimensions in fine address units (pixels) are listed in the table below.

Table 3–3: Character Dimensions in Fine Address Units

	SIZE n			SIZE n,n	
n	width	default y-size	height	width	height
0	9	0	0	9	0
1	9	2	20	9	10
2	18	3	30	18	20
3	27	5	50	27	30
4	36	6	60	36	40
5	45	8	80	45	50
6	54	9	90	54	60
7	63	11	110	63	70
8	72	12	120	72	80
9	81	14	140	81	90
10	90	15	150	90	100
11	99	17	170	99	110
12	108	18	180	108	120
13	117	20	200	117	130
14	126	21	210	126	140
15	135	23	230	135	150
16	144	24	240	144	160
17				|	170
18				|	180
19				|	190
20				|	200
21				|	210
22				|	220
23				|	230
24				|	240
25				144	250

Remember that row and column addresses specify locations that are the right size for size 1 characters. After you select a different size, the horizontal spacing and the vertical spacing for the next text or text block is appropriate for the new size. When you use the AT instruction to set a new current location, you must calculate the location so that you do not write over text you have already displayed.

Figure 3–7 shows text in three sizes. The grid on this figure marks every 10 columns and every 5 rows.

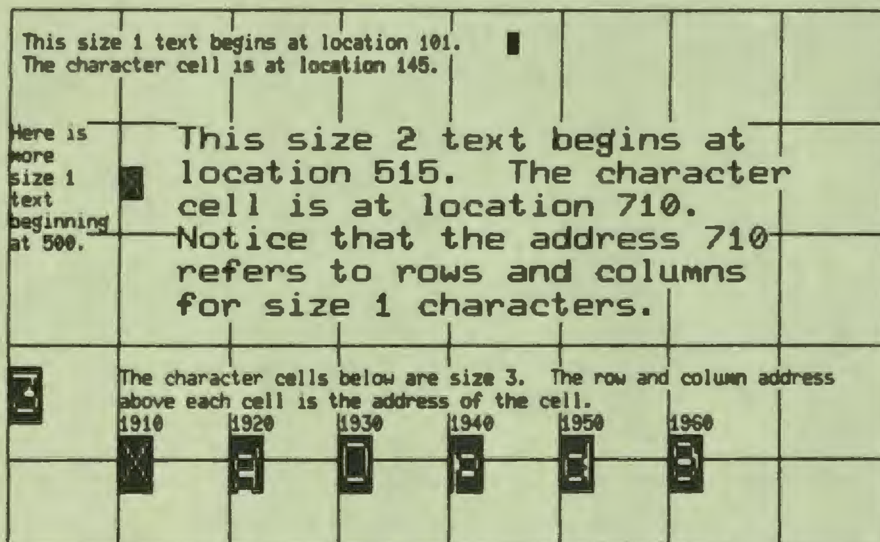


Figure 3–7
Changing Text Size

Look at the size 1 text and the size 2 text in the center of the figure. Three rows of size 1 text require the same vertical space as two rows of size 2 text. This relationship can be calculated from Table 3–3. Each size 1 character is 20 units high. Each size 2 character is 30 units high. Multiplying the number of units per row by the number of rows results in 60 units in both cases.

The inverse characters at the bottom of Figure 3–7 are size 3. Size 3 characters are 50 units high. These characters are displayed at row 19. To display more characters below them, you need to calculate the row. The rows in row-and-column addresses are 20 units high, so each size 3 character uses two and one-half rows. You have two choices. You can use three rows and specify row 22, or you can calculate the address in fine coordinates.

The row-and-column address of the size 3 character X is 1910. In fine coordinates, the address of the top left corner of the character cell is 90,380. The x-coordinate equals the column * 9. The y-coordinate equals the row * 20.

Size 3 characters are 27 units wide. To display the next character to the right of the X, add 27 to the x-coordinate. The address is 117,380. Size 3 characters are 50 units high. To display the next character below the X, add 50 to the y-coordinate. The address is 90,430.

USING COLOR

Two current colors are always used for displaying text and graphics on the monitor. The color of the screen is the background color, and the writing color is the foreground color.

The default background color is DARK. The instruction BCOLOR changes the background color. The entire screen changes color as soon as BCOLOR is executed.

The instruction FCOLOR sets a new current foreground color. After an FCOLOR instruction is issued, the text and graphics that follow are displayed on the screen in the current foreground color. Text and graphics already on the screen do not change color.

At lesson startup, the VAX DAL color system is initialized with the eight colors listed below.

Color Name	Number
DARK	0
BLUE	1
RED	2
MAGENTA	3
CYAN	4
GREEN	5
YELLOW	6
WHITE	7

Each of the DAL-provided colors can be specified either by name or by number in an FCOLOR or BCOLOR instruction. If the lesson is displayed on a black-and-white monitor, the colors and numbers select shades of gray from darkest to lightest.

Chapter 12 in this manual discusses how to use VAX DAL instructions to take full advantage of the color capabilities of a terminal. This may involve using as many as 56 additional colors.

The CCOLOR instruction clears DAL's internal color table. The DAL color table limits the number of colors available for use to the number of colors that can be supported on the terminal screen at the same time. CCOLOR is useful for terminals that can employ only a limited number of colors simultaneously. If you specify three writing colors and a background color on a four-color terminal, and you want to use a color you have not already specified, you can use the CCOLOR instruction or an FCOLOR instruction with a table_slot_number argument to stop using one of the current colors and start using the new color. To stop using a color, execute a CCOLOR instruction with the same parameters that the BCOLOR and FCOLOR instructions use.

CCOLOR also accepts the keyword ALL to clear the entire color table. After a CCOLOR ALL is issued, there is no current writing color. A new color must be specified; otherwise, results are unpredictable.

The color DARK specifies that the screen is not illuminated. On both black-and-white and color monitors, the student can alter the screen from black to gray with the brightness and contrast controls on the monitor.

DISPLAY MODES

The current display mode affects the way text and graphics are displayed on the screen.

The MODE instruction selects the current display mode. The argument to the MODE instruction is a keyword that describes the mode. Three mutually exclusive pairs of keywords are explained here. Two other modes are explained in Chapter 10.

Table 3–4 summarizes three mutually exclusive pairs of keywords used as arguments to the MODE instruction.

Table 3–4: Mode Keywords

Keyword	Description
NORMAL/INVERSE	The keyword NORMAL displays graphics and the dots for characters in the foreground color, with the rest of the character cell in the background color. The keyword INVERSE displays graphics and the dots for characters in the background color, with the rest of the character cell in the foreground color. The keyword INVERSE must be used with care with graphics. In some cases, graphics are not displayed when the INVERSE mode is in effect. The default is NORMAL.
OVERLAY/REPLACE	The keyword OVERLAY displays text and graphics over anything already displayed. The keyword REPLACE first erases the dots for the new graphics or text, then displays the new graphics or text. The default is OVERLAY.
FIXED/BLINK	The keyword BLINK causes text and graphics to alternate between the foreground color and DARK. The keyword FIXED does not blink the display. The default is FIXED. Note that blink mode does not work on all types of terminals.

In addition to using the MODE instruction with keywords, authors can use system constants with the MODE instruction to change display modes. For example, a MODE INVERSE instruction has the same effect as a MODE M_INVERSE instruction. The following system constants can be used instead of the MODE keywords:

Table 3–5: Mode System Constants

Keyword	Constant
NORMAL	M_Normal
INVERSE	M_Inverse
OVERLAY	M_Overlay
REPLACE	M_Replace
FIXED	M_Fixed
BLINK	M_Blink

All figures in this chapter except 3–2 and 3–5 show both inverse and normal modes.

The difference between the grids in Figures 3–1, 3–2, and 3–3 and the grids in Figures 3–6 and 3–7 illustrate overlay mode and replace mode.

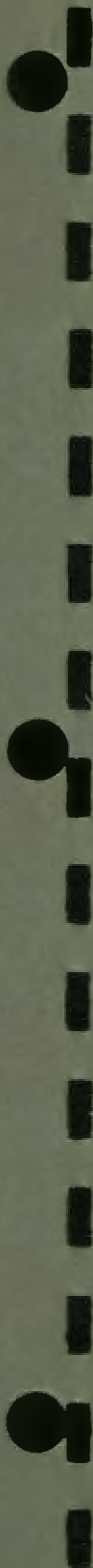
In Figures 3–1, 3–2, and 3–3, the grids are displayed first, then the rest of the illustration is drawn in overlay mode. Notice that the characters and the grid are both displayed, even when the characters are on the grid lines. In Figures 3–6 and 3–7, the text is displayed in replace mode. Notice that the grid is erased under the character cells.

Figure 3–4 also shows how overlay mode can be used to write a circumflex over a word.

There are three current modes selected by one keyword from each of the pairs listed in Table 3–4. Displays are in fixed, normal, and overlay modes, or blink, normal, and replace modes, or any other combination of one mode from each of the mutually exclusive pairs. Remember that blink mode does not work on every type of terminal.

4

Default Response Judging



4

Default Response Judging

With the AT and WRITE instructions explained in Chapter 3, you can display explanations and questions for the student to answer. This chapter explains the instructions that read the student's response and judge it. This chapter discusses default response judging. The complete lesson in Chapter 5 and all of Chapter 8 discuss modifications to the default response-judging process.

THE RESPONSE-JUDGING BLOCK

A response-judging block reads the student's response from the keyboard and judges it.

In the response-judging block, you specify as many anticipated right answers and wrong answers as are appropriate for the question. After each right or wrong answer, you can include instructions that are executed only if the response matches that answer. These are called response-contingent instructions. Then you indicate the end of the block.

The QUERY instruction begins a response-judging block; the ENDQ instruction ends it. The RIGHT instruction specifies right answers. The WRONG instruction specifies wrong answers.

The following instructions show a response-judging block.

```
QUERY
RIGHT     Harriet Beecher Stowe
.         response-contingent instructions
RIGHT     Louisa May Alcott
.         response-contingent instructions
WRONG     Aimee Semple McPherson
.         response-contingent instructions
ENDQ
next instruction
```

Only one response-judging block is allowed at lesson level and in each unit.

QUERY

When the QUERY instruction is executed, the lesson displays the default prompt character, the right angle bracket (>), then pauses until the student types a response. The response ends with the default delimit character (RET), which is transmitted by the RETURN key.

The location of the prompt character depends on the argument to the QUERY instruction. When QUERY has no argument, the prompt character is displayed at the left margin and one line below the last displayed text. An asterisk (*) as the argument displays the prompt character one space to the right of the last displayed text. A screen address as the argument displays the prompt character at that address.

Figure 4–1 shows the arguments to QUERY and their results.

```
Here is a question.                      Here is another question.
The prompt character                     The instruction
below is displayed by                    QUERY   *
the instruction QUERY.                   displays the prompt
>student's response                      character to the right>response




              Here is the third question.  The
              prompt character is displayed by
              the instruction QUERY  1925

                   >another response
```

MR-S-2118-82

Figure 4–1
The QUERY Instruction

Figure 4–1 also shows responses. Each character of the response is echoed as the student types it. The echoed response begins in the character position to the right of the prompt character. The prompt character and the student's response are displayed in the current text size.

When the student presses the RETURN key, the response is read.

RIGHT and WRONG

The RIGHT instruction specifies anticipated right answers. The WRONG instruction specifies anticipated wrong answers.

The response is compared to each specified right and wrong answer in the order they are specified. The first match determines what happens next, and all following RIGHT and WRONG instructions are ignored. If the response matches a right answer, it is scored with a value of 1. Any response-contingent instructions following the RIGHT instruction are executed. The lesson goes to the ENDQ instruction and continues from that point.

If the response matches a wrong answer, any response-contingent instructions following the WRONG instruction are executed. Then the lesson erases the student's old response and returns to the QUERY instruction. The QUERY instruction displays the prompt character and waits for a new response. Judging is repeated.

If the response matches none of the anticipated answers, the response is judged wrong. The lesson erases the old response and returns to the QUERY instruction.

By default, a response-judging block ends only when the student enters a response judged right.

Figure 4–2 shows the transfer of control after responses judged right and responses judged wrong.

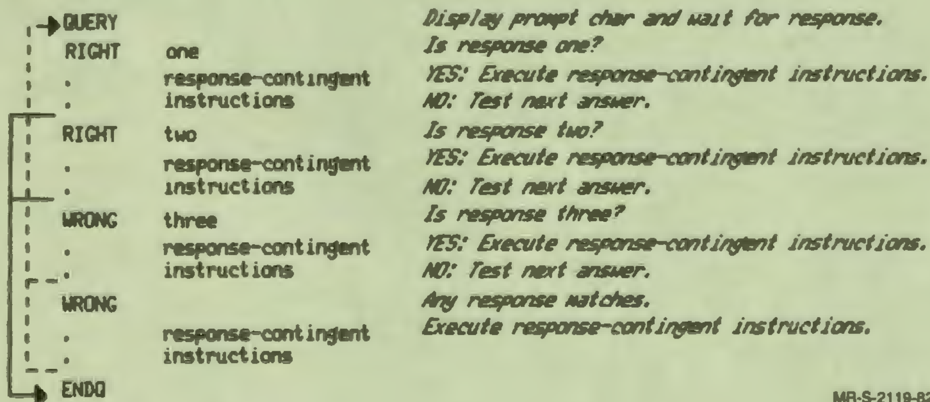


Figure 4–2
Response-Judging Block

Several RIGHT instructions and WRONG instructions can be included in one response-judging block. Several answers can also be specified with one RIGHT or WRONG instruction.

The response-judging block above shows one argument to each of the RIGHT instructions and to one WRONG instruction. The other WRONG instruction has no argument. The response "one" matches the answer specified by the first RIGHT instruction. The response "two" matches the answer specified by the second RIGHT instruction.

Because each RIGHT and WRONG instruction can be followed by response-contingent instructions that are executed only when the response matches that specified answer, specifying each anticipated answer separately allows the feedback to be different for each response. Several answers can also be specified with one RIGHT or WRONG instruction. The answers are separated by a vertical bar (|) as shown below.

```
RIGHT    Plato | Aristotle | Socrates
.        response-contingent instructions
```

In this case, the same response-contingent instructions are executed if the response matches any one of the three answers.

The RIGHT and WRONG instructions can have no argument. In this case, any response matches. It is often useful to use WRONG with no argument at the end of a response-judging block. Then feedback can be displayed for a response that did not match any of the anticipated answers.

DISPLAYING FEEDBACK

DAL lessons do not display any feedback automatically, nor do they pause in a response-judging block except at the QUERY instruction.

Two instructions provide simple feedback. The MARKUP instruction displays the word OK after a response is judged right or the word NO after a response is judged wrong. The PAUSE instruction stops execution of the lesson until the RETURN key is pressed so the student has time to read the feedback.

The following response-judging block uses MARKUP and PAUSE.

```
QUERY
RIGHT    Socrates | Aristotle | Plato
.        MARKUP
.        PAUSE
WRONG
.        MARKUP
.        PAUSE
ENDQ
```

If the student's response is Socrates, Aristotle, or Plato, the word OK is displayed two spaces to the right of the response. The lesson pauses until the student presses the RETURN key, then continues at the ENDQ instruction.

If the student's response is anything else, the word NO is displayed. The lesson pauses until the student presses the RETURN key. The response is judged wrong, so the lesson erases the response and the word NO. Control returns to the QUERY instruction, which displays the prompt character and pauses until the student enters a new response.

The PAUSE instruction can be used any place in the lesson. It is not restricted to response-judging blocks. All answers do not require the PAUSE instruction. After a response matches a right answer, instructions following the ENDQ instruction can display feedback. Because of the default processing, the PAUSE instruction is usually included in the response-contingent instructions after WRONG. Figure 4–3 shows the results of the MARKUP instruction.

```
Name a Greek philosopher.
>Willain James  NO


Name a Greek philosopher.
>Aristotle  OK
```

MR-S-2120-82

Figure 4–3
Markup

You can display any feedback you want by using AT and WRITE instructions after each RIGHT and WRONG instruction. In this case, you must also erase the feedback.

The following response-judging block displays the student's response as part of the feedback for a wrong answer.

```
QUERY
RIGHT     Aristotle | Plato | Socrates
.         MARKUP
.         PAUSE
WRONG
.         AT        1510
.         WRITE     <<S,RESPONSE>> is not a Greek philosopher.
                    Press RETURN and try again.
.         PAUSE
.         ERASE     1510;2480
ENDQ
```

The system variable RESPONSE contains the student's response. If the response is wrong, the preceding instructions display the response and the rest of the text, then pause. When the student presses the RETURN key, the lesson erases the screen from row 15, column 10 to row 24, column 80.

Because the student's response matched a wrong answer, the response is erased automatically. The prompt is displayed again, and the lesson waits for a new response. If the student's new response is not one of the three names specified as right answers, the response is erased, the prompt is displayed, and so on.

After the student enters a response, the lesson changes the current location. When WRITE is used without a preceding AT in response-contingent instructions, the text begins two rows below the first character of the response. The spacing is appropriate for the current text size.

SPECIFICATIONS FOR MATCHING

The student's response and the specified answer do not need to be identical to be judged as matching.

By default, differences in uppercase and lowercase characters are ignored. Most punctuation marks in the specified answer and the response are ignored. The exceptions are the apostrophe, the hyphen, the dollar sign, and the underscore. These punctuation marks are considered part of the word. A spelling tolerance test is applied to each word in the response, so that mistyped or misspelled responses match if they meet the requirements of the tolerance test.

By default,the response cannot contain fewer words, more words, or the same words in a different order.

The instruction line below specifies a right answer.

```
RIGHT     Louisa May Alcott
```

The following responses match, although they are not identical:

Louise May Alcot (misspelled)
louisa May alcott (different uppercase letters)
Louise may. Alcott (misspelled and has punctuation)

The following responses do not match:

Mrs. Louise May Alcott (extra word)
May Louisa Alcott (words in different order)
Louisa Alcott (missing word)

The SPECS instruction can be used to modify default response judging. See Chapter 8 in this guide for more information about the SPECS instruction.

5

Creating a Simple Lesson

5

Creating a Simple Lesson

This chapter presents one lesson written in VAX DAL. The chapter explains the division of a lesson into lesson-level instructions and units, and traces the control logic that determines the order in which instructions are executed. The chapter also discusses defining variables, assigning values to the variables, and using system variables.

The lesson uses the graphics instructions explained in Chapter 3 and the response-judging instruction explained in Chapter 4. The specific uses of these instructions in the sample lesson are explained.

The lesson used throughout this chapter as an example is named Multiply. This lesson lets students choose whether to review multiplication tables or to practice multiplication problems. The complete listing of the lesson is in Appendix F.

You should execute the lesson before reading this chapter. This will make the information presented here easier to understand. You will be able to compare the displays to the VAX DAL instructions that create them, and follow the sequence of instructions in the lesson by seeing what happens on the screen.

Complete lesson units are listed in the text of this chapter, and each unit includes all the instructions needed for the function of the unit. As you read the chapter, there may at first be a number of instructions in the lesson units that you do not recognize and do not understand. This is because the chapter is organized by topics to help you learn VAX DAL, and only those instructions that illustrate the topic being explained are discussed in detail. As you read and accumulate information, more of the instructions in each unit will become familiar.

STRUCTURE OF A LESSON

The lesson Multiply lets students choose whether they want to review their multiplication tables or practice multiplication problems. Students who choose to review can then choose to review all the multiplication tables or just one. Students who choose to practice see 25 multiplication problems and give answers. The lesson then displays the score for the set of problems. After students have finished reviewing or practicing, they can choose again. The lesson ends when students choose to quit.

When students practice more than once, the lesson displays the score for the current set of problems and the score for the previous set.

Lessons are generally divided into lesson-level instructions and a number of units. Each unit consists of instructions to perform one function, such as displaying a menu or displaying and judging a problem.

Dividing the lesson into units has two advantages:

- A series of instructions that are needed a number of times in the lesson can be written once as a unit. The unit is then called by a DO instruction at a higher level. All the instructions in the unit are executed, then control returns to the instruction following the DO instruction. Since units are written once but can be executed any number of times, units save time and help prevent mistakes.
- Units clarify the structure of a lesson and simplify writing it by dividing the lesson into smaller pieces.

When writing a lesson, place lesson-level instructions at the beginning. The first instruction must be the LESSON instruction. Lesson level ends at the first UNIT instruction. Each unit begins with a UNIT instruction and ends at the next UNIT instruction. Units can be written in any order.

When students take the lesson, execution begins at the LESSON instruction and continues through the lesson-level instructions. Units are executed by DO instructions at lesson level or in other units. A DO instruction in one module can execute a unit in a second, separately compiled module only if the modules are linked together after they are compiled (see Chapter 6).

Planning the Lesson

The lesson Multiply divides structurally into three units: one for practicing, one for reviewing, and one for ending the lesson. The names of the units are practice, review, and quit. Since the only function of the unit quit is to display a message, quit is not divided into other units.

```
LESSON   multiply
DEFINE   go_on:BOOLEAN                    $$ True until student quits.
ASSIGN   go_on := TRUE
DEFINE   done_once:BOOLEAN                $$ Used in unit practice.
DEFINE   x,y,z:INTEGER                    $$ Used for all multiplication.

DO       intro1                           $$ Display title page.
SCORE    FALSE                            $$ Turn off scoring.

; Set up loop. Lesson returns here after each practice or
; review. Loop is broken and lesson ends when student chooses to quit.
LOOP     go_on                            $$ Begin loop.
.        DO       menu                    $$ Display instructions and choices.
.        QUERY
.        RIGHT    practice
.        .        DO       practice
.        RIGHT    quit
.        .        DO       quit
.        .        ASSIGN   go_on := FALSE
.        RIGHT    REVIEW
.        .        DO       review
.        WRONG                            $$ Anything else is wrong.
.        .        SIZE     1              $$ Display error message
.        .        WRITE    YOU MUST TYPE ONE OF THE WORDS ABOVE
.        .        SIZE     2              $$ and return to beginning
.        ENDQ                             $$ of QUERY block.
ENDLOOP                                   $$ end of loop

; End of lesson level.
```

The first instruction, as required, is LESSON. The argument to LESSON is the name of the lesson.

Lesson-Level Variables

The DEFINE instruction defines the names and data types of variables. The syntax is:

```
DEFINE   variable_name{,...}:data_type{,usage characteristic}
```

Chapter 2 explains the data types and usage characteristics.

Variables defined at lesson level can be used at lesson level and in any unit. The five lesson-level variables are:

- go_on

  The variable go_on is used only at lesson level. As its name suggests, the variable is used to test whether the student wants to go on. The variable go_on is defined as Boolean, which means that it has one of two possible values, true or false. The instruction ASSIGN go_on := TRUE assigns the initial value true to go_on.

- done_once

  The Boolean variable done_once is used in the unit practice to test whether the student has practiced once. When Boolean variables are defined, they are assigned the value FALSE. FALSE is the right initial value for done_once. Although this variable is used in only one unit, it is defined at lesson level so that it is initialized only once. The reason this is important is explained when the control logic for the unit practice is explained.

- x,y,z

  These three integer variables are used in the units review and practice. At any time, they contain the values for the current multiplication problem or the current times table display. All problems and displays correspond to the form x * y = z.

  Integer variables (and real variables) are initialized as zero.

Lesson-Level Control Logic

The next instruction, DO intro1, executes the instructions in the unit named intro1. This unit displays the title page for the lesson, with the border around the outside and the word MULTIPLICATION. After the last instruction in the unit intro1 is executed, control returns to the instruction following the DO instruction.

When a lesson begins, the responses to all queries are scored. In Multiply, however, only the practice problems should be scored. The SCORE instruction determines whether responses are scored; its argument is a Boolean expression. SCORE FALSE turns off scoring. SCORE TRUE turns it on. The SCORE FALSE instruction, then, stops scoring.

Because the unit practice controls the practice problems, the unit also turns scoring on and off.

The unit review divides into two units, the unit revone to display one times table, and the unit revall to repeat the display 12 times. These units have both of the advantages listed above. The division clarifies the structure of the unit review. The units revone and revall also are repetitive. Unit review calls unit revone when the student chooses to review one times table. Unit review calls unit revall when the student chooses to review all the times tables. Unit revall then calls unit revone 12 times.

The multiplication problems are also repetitive. Unit prac1 displays one problem, judges the student's response, and displays feedback. Unit practice calls unit prac1 25 times.

Figure 5–1 shows the structure of the lesson described above. Lesson level, unit prac1, and unit review use response-judging blocks. The QUERY instructions and the DO instructions to call other units are shown in Figure 5–1. Summaries of the other functions the lesson and the units must perform are shown in italics.

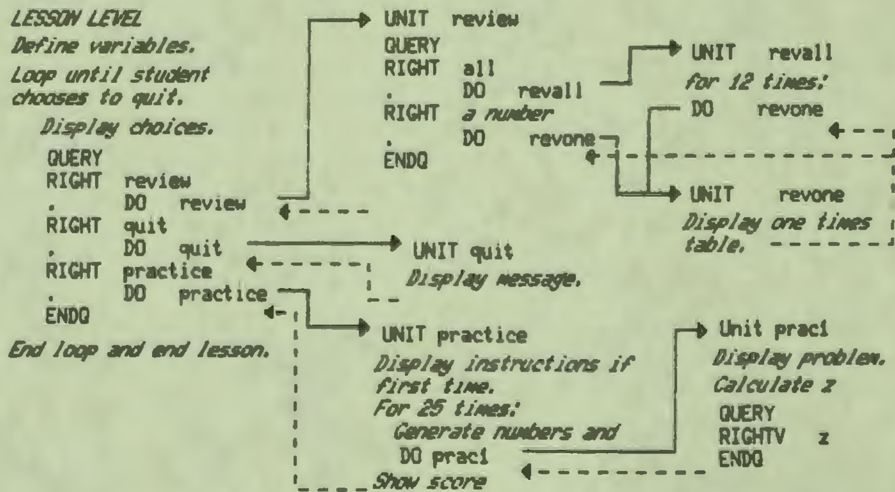


Figure 5–1
Lesson Structure

Planning Units

One restriction must be considered in planning units. Each unit can contain only one response-judging block. The units planned for Multiply meet this restriction. There is only one QUERY instruction in lesson level, one in the unit review, and one in the unit prac1.

The subject matter of a lesson generally suggests a natural division into units. For the lesson Multiply, these units are shown by name in Figure 5–1. The unit practice calls two other units, instruc and shoscore. The unit instruc displays the instructions the first time the student chooses to practice, and calls unit prac1 to display and judge a sample problem. The unit practice calls shoscore to display the students' scores.

During lesson planning it becomes clear which additional units might be useful. For example, lesson-level instructions can be simplified here by making a unit of the menu that shows the student's three main choices.

Because the instructions in return perform a function that is repeated frequently, the author is able to write the lesson more efficiently by including these instructions in one unit. Students taking the lesson need time to read the screen, and must be able to indicate when they are ready to continue. The unit return displays the message PRESS RETURN, pauses, and erases the screen. This unit is executed after each times table, after the feedback for each problem, after the display of instructions, and after the display of scores.

The lesson Multiply also displays a title page. The title page is in the unit intro1.

The structure of the lesson shown in Figure 5–1 suggests other units as well. Some of the summaries shown in italics, for example, could be organized into individual units.

LESSON-LEVEL INSTRUCTIONS

The LESSON instruction is the first element in a lesson. Its argument is the lesson name.

Lesson level ends at the first UNIT instruction. After the last lesson-level instruction is executed, the lesson ends.

The following instructions are the lesson-level instructions for Multiply.

The rest of the instructions are part of the control logic for the lesson. The requirements are that students can choose to review or to practice, and that they can do this until they decide to quit. In Figure 5–1, the control logic at lesson level is described as a loop. The instruction LOOP begins the loop. The instruction ENDLOOP ends it.

The syntax of the LOOP structure is:

```
LOOP      Boolean expression
          Instructions
ENDLOOP
```

When a LOOP instruction is executed, its argument is tested. If the argument is true, the instructions between LOOP and ENDLOOP are executed. At the ENDLOOP instruction, control is passed back to the LOOP instruction; its argument is tested again, and so on, until the argument is false. When the argument is false, control is passed to the instruction after ENDLOOP. At some point in the loop, the value of the expression being tested must change. Otherwise, the loop never ends.

In this LOOP instruction, the argument is the Boolean variable go_on. The first time the LOOP instruction is executed, the Boolean variable go_on is true because the ASSIGN instruction above assigned the value TRUE to go_on. So the instructions between LOOP and ENDLOOP are executed at least once.

Figure 5–2 shows the LOOP instruction and the way this loop controls execution of the other units in the lesson.

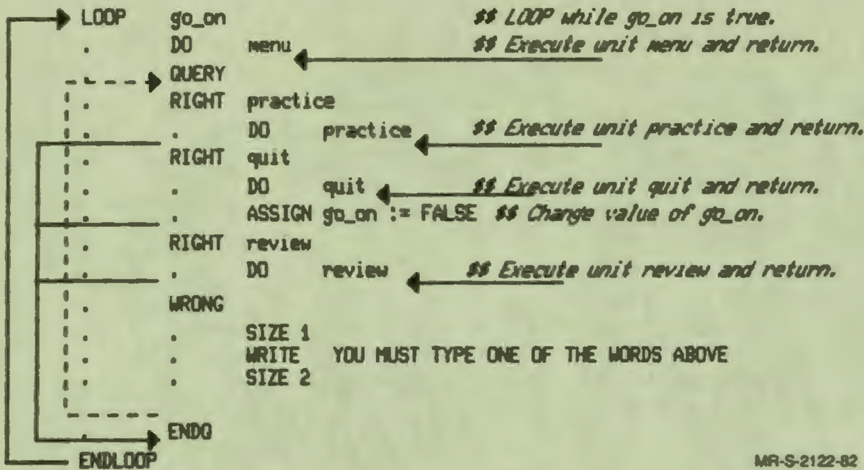


Figure 5–2
Lesson-level Control Logic

One of the specified right answers in the lesson-level response-judging block is quit. The response-contingent instruction ASSIGN go_on := FALSE changes the value of the variable that controls the loop. When students choose to quit, the loop ends. Because the instruction ENDLOOP is also the last instruction at lesson level, the lesson ends.

In the loop are a DO instruction and a response-judging block. The DO menu instruction calls the unit menu, which displays the three choices.

The response-judging block beginning with QUERY and ending with ENDQ reads and judges the response. If the response is practice, the response-contingent instruction DO practice calls the unit practice. If the response is review, the response-contingent instruction DO review calls the unit review. If the response is quit, the response-contingent instructions call the unit quit, then assign a new value to the variable go_on. After any response that matches one of the right answers, the rest of the specified answers are ignored. The lesson goes to the instruction following ENDQ. The ENDLOOP instruction returns to the LOOP instruction.

The WRONG instruction in the response-judging block is an error-checking instruction. When either RIGHT or WRONG has no argument, any response matches.

Because of the system defaults, students can use uppercase or lowercase letters and can misspell any of the responses. Students who read the instructions will probably type "practice" or "review" or "quit", and there will be no attempt to match the argument to WRONG. If, however, students type anything except one of these three words, the response matches the argument to WRONG. The response-contingent instructions display a message explaining what to do, and the lesson erases the response. Because the response matches an answer specified as wrong, the lesson returns to the QUERY instruction and waits for another response.

DISPLAYING TEXT

The AT and WRITE instructions are discussed in Chapter 3. This section shows more examples of these instructions and explains how to change the prompt character.

Changing Character Sizes

There are three character sizes in this lesson. Because the lesson is for fairly young children, size 1 is used very little. The review multiplication tables are size 2. This is bigger than size 1 and easier to read. Size 3 is too big for a complete multiplication table to fit on the screen at once. Size 3 is used for the practice problems.

The explanatory text in the unit menu is a combination of sizes. This is a compromise between large characters and the information that needs to be shown at one time.

Here is the unit menu, which displays the students' choices. Figure 5–3 shows the display.

```
UNIT    menu                 $$ Display three main choices.
FCOLOR  RED
ERASE
BOX     0,0;0.999,0.999      $$ For manual illustrations.
AT      210
SIZE    2
WRITE   This lesson lets you review your
        multiplication tables or practice
        some multiplication problems.
AT      820
WRITE   Do you want to
AT      1125
WRITE   PRACTICE
SIZE    1
```

```
AT      1332
WRITE   OR
SIZE    2
AT      1427
WRITE   REVIEW
SIZE    1
AT      1632
WRITE   OR
SIZE    2
AT      1729
WRITE   QUIT
```

```
This lesson lets you review your
multiplication tables or practice
some multiplication problems.

     Do you want to

        PRACTICE
            OR
         REVIEW
            OR
          QUIT
          >practive
```

MR-S-2123-82

Figure 5–3
The Unit Menu

Figure 5–3 also shows a prompt character and a response. The unit menu is executed by the DO menu instruction in the lesson-level loop. The prompt character is displayed by the QUERY instruction following the DO menu instruction at lesson level. The default specifications for matching responses and specified answers are in effect, so this response matches the answer practice even though it is misspelled.

The first instruction in the unit is the UNIT instruction. This instruction marks the beginning of a unit; the argument is the unit name.

The ERASE instruction erases the screen. The FCOLOR instruction changes the current foreground color to red. The BOX instruction outlines the screen.

The SIZE 2 instruction changes the current text size to size 2, and the AT 210 instruction sets the current location. The first three lines of text begin at row-and-column address 210.

Because the first three lines are a text block displayed by one WRITE instruction, both the column spacing and the row spacing are appropriate for size 2 characters.

The argument to the next AT instruction specifies the address in row-and-column coordinates. Row-and-column coordinates specify units the right size for size 1 characters. Because the text is size 2, the address for the next line of text must be calculated to indent the line and space the rows properly.

Each size 2 character is 30 units high; three rows therefore require 90 units. Since each row in row-and-column units is 20 units high, the three lines already displayed use four and one-half rows. The lines started at row 2. Specifying row 8 for the next line leaves one and one-half rows between the lines of text.

Each size 2 character is 18 units wide. Each column is 9 units wide. Selecting column 20 for the next line indents the line an additional 10 columns, which is equivalent to five size 2 characters.

The row-and-column addresses for the rest of the display are calculated the same way.

The unit menu ends after displaying the word quit. Control returns to lesson level. The next instruction following DO menu is QUERY. The current text size is still size 2. The prompt character displayed by the QUERY instruction and the student's response, therefore, are size 2.

The response-judging block in the unit review shows another use of text in different sizes.

The unit review follows.

```
UNIT     review                    $$ Select to review one or all tables.
ERASE
BOX      0,0;0.999,0.999           $$ For manual illustrations.
SIZE     2
AT       510
WRITE    Type ALL to review all
         the multiplication tables.

         Type a number to review
         that multiplication table.
QUERY
RIGHT    all
.        ERASE
.        BOX      0,0;0.999,0.999     $$ For manual illustrations.
.        DO       revall              $$ Review all.
RIGHT    1|2|3|4|5|6|7|8|9|10|11|12
.        ASSIGN   x := NUMBER(RESPONSE)
.        ERASE
.        BOX      0,0;0.999,0.999     $$ For manual illustrations.
.        DO       revone              $$ Review chosen one.
WRONG
.        SIZE     1
.        WRITE    You must type ALL or a number from 1 to 12.
                  Try again.
.        SIZE     2
ENDQ
```

Figure 5–4 shows the text displayed by the unit review with a student's response typed in. The response shown does not match any specified right answer. The WRONG instruction in the response-judging block has no argument. Any response that did not match one of the specified right answers matches here. The response-contingent instructions display an error message.

```
Type ALL to review all
the multiplication tables.

Type a number to review
that multiplication table.
>

  You must type ALL or a number from 1 to 12.
  Try again.
```

MR-S-2124-82

Figure 5–4
The Unit Review

The error message is displayed at the default location for a response-contingent WRITE instruction. This line begins two rows below the last line written (which is the student's response), and in the same column as the first character of the response.

The instruction SIZE 1 sets the current text size to size 1 before the line is displayed. The DO return instruction executes the unit return. The SIZE 2 instruction sets the current text size back to size 2. After executing the response-contingent instructions for a response judged wrong, the lesson erases the response and displays the prompt character again. The prompt character and the response are displayed in the current text size. Because the response-contingent SIZE 1 changed the current text size, the instruction SIZE 2 is necessary. Without this instruction, the prompt character and the next response would be displayed in size 1.

Changing the System Prompt Character

The instructions below are from the unit prac1, and display one multiplication problem. The WRITE instruction displays the two variables x and y in string format. (Displaying variables is explained in Chapter 3.)

```
PROMPT "= "
AT     820
WRITE  <<S,x>> x <<S,y>>
QUERY  *
```

Figure 5–5 shows a problem.

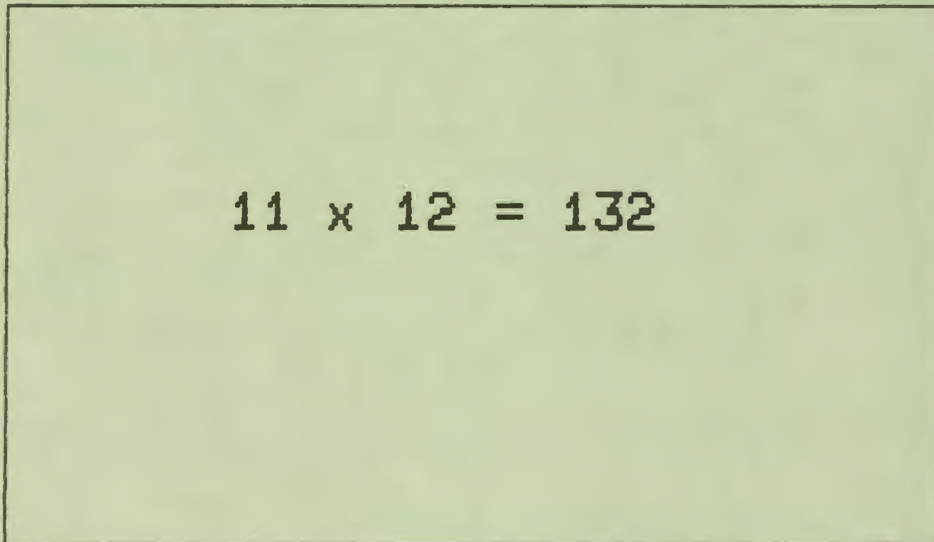


Figure 5–5
Changing the Prompt Character

The instruction PROMPT specifies the prompt character. The argument can be a string constant, variable, or expression. The argument in this case is a string constant, and as such is enclosed in double quotation marks ("). There is a space character after the equal sign (=).

After this instruction is executed, the QUERY instruction displays an equal sign and a space. Another PROMPT instruction later in the unit changes the prompt character back to the angle bracket (>).

The asterisk (*) argument to the QUERY instruction displays the prompt one space to the right of the last text displayed. There is a space after the angle brackets for the variable y. This space does not show in the instructions, but it does show in Figure 5–5.

The WRITE instruction displays 11 x 12 and the space between the 12 and the equal sign. The QUERY instruction displays the equal sign and the space following it. Changing the prompt character and changing its default location make the display look like a multiplication problem.

Using Color

This lesson uses several foreground colors. Some of the color is decorative and some is functional. The most obvious functional use of color is in the unit prac1. The multiplication problem and the students' response are blue. If the response is right, the unit displays "you're right" and repeats the problem and answer. If the response is wrong, the unit displays "no, the answer is" and repeats the problem with the right answer. The two kinds of feedback look much alike, and color is used to distinguish them. If the response is right, the feedback is blue. If it is wrong, however, the feedback is red.

Changing Display Modes

The unit return shows another text writing effect, the inverse mode. In inverse mode, the foreground and background colors for the text cells are changed. Here is the unit.

```
UNIT    return           $$ Display press return and
SIZE    1                $$ wait at pause.
AT      2060
WRITE   PRESS
MODE    INVERSE
WRITE   RETURN
PAUSE
ERASE
MODE    NORMAL
BOX     0,0;0.999,0.999
```

The default display mode is normal, and no instructions any place else in the lesson change it. The word PRESS, therefore, is displayed in normal mode. This word is followed by a space that does not show in the example. The instruction MODE INVERSE changes the current mode to inverse, then the word RETURN is displayed. For this word, the foreground and background colors of the dots for the character and the rest of the character cell are reversed.

The PAUSE instruction stops execution of the lesson until the student presses the RETURN key. Then the instruction MODE NORMAL changes the current display mode back to normal.

Displaying Variables

The units revone and prac1 display the contents of variables to show students the times tables and the multiplication problems.

Three variables, x, y, and z, are defined at lesson level. At any time, these variables contain the numbers for the current display.

In unit prac1, the system prompt character is changed to an equal sign followed by a space. The WRITE instruction that displays one problem is:

```
WRITE     <<S,x>> x <<S,y>>
```

The prompt character follows the display. Figure 5–5 shows one problem after the student has typed the response.

The instruction that displays one line of a times table is:

```
WRITC     <<T,x,2,0>> x <<T,y,2,0>> = <<T,z,3,0>>
```

A complete times table is shown in Figure 5–6.

```
9 x  0 =   0
9 x  1 =   9
9 x  2 =  18
9 x  3 =  27
9 x  4 =  36
9 x  5 =  45
9 x  6 =  54
9 x  7 =  63
9 x  8 =  72
9 x  9 =  81
9 x 10 =  90
9 x 11 =  99
9 x 12 = 108         PRESS RETURN
```

MR-S-2126-82

Figure 5–6
Displaying Variables

The numbers for the problem are displayed in string format. The times table is displayed in tabular format. The different formats are needed because of the columns in the times tables.

The double angle brackets are the syntax for displaying the contents of a variable. The argument S before the variable name selects string format. For integer variables, string format converts the number to digits and displays the digits beginning at the current location. Although the problems use both 1-digit numbers and 2-digit numbers, string format spaces the number appropriately.

The problem 1(SP)x(SP)1(SP) requires six positions. The problem 12(SP)x(SP)12(SP) requires eight positions. Since only one problem is displayed at a time, this difference is acceptable. In the times tables, however, 13 problems are displayed at the same time. In string format, the multiplication signs and the equal signs would not line up.

The tabular format is a better choice for the multiplication tables.

The T argument before the variable name selects tabular format. Tabular format requires two arguments after the variable name. The first argument defines the total number of positions required by the variable. For the variables x and y, this number is 2. These variables are always a number from 0 to 12, and never require more than two positions. For the variable z, this number is 3. The value of z in some times tables is over 100; therefore, z requires three positions.

The last argument in tabular format is the number of positions to the right of the decimal point. Since these variables are all integers, this argument is zero.

You can see how this works in Figure 5–6.

The WRITC instruction is a variation of the WRITE instruction. The instruction WRITC, which can also be spelled WRITEC, inserts a carriage return at the beginning of the line. For an explanation of how the values of the variables x, y, and z are changed for the times tables and why the WRITC instruction is used, see the section of this chapter that deals with the FOR instruction.

JUDGING THE STUDENTS' RESPONSES

The lesson Multiply contains three response-judging blocks: one at lesson level, one in the unit review, and one in the unit prac1.

Specifying Several Answers

The response-judging block at lesson level executes one of three units, depending on the student's response. The response-judging block in the unit review is similar, but also specifies a number of right answers with one RIGHT instruction.

The complete unit is shown above. The response-judging block is repeated below.

```
QUERY
RIGHT   all
.       ERASE
.       BOX     0,0;0.999,0.999     $$ For manual illustrations.
.       DO      revall              $$ Review all.
RIGHT   1|2|3|4|5|6|7|8|9|10|11|12
.       ASSIGN  x := NUMBER(RESPONSE)
.       ERASE
.       BOX     0,0;0.999,0.999     $$ For manual illustrations.
.       DO      revone              $$ Review chosen one.
WRONG
.       SIZE    1
.       WRITE   You must type ALL or a number from 1 to 12.
                Try again.
.       SIZE    2
ENDQ
```

The ENDQ instruction is the last instruction in the unit.

The second RIGHT instruction specifies 12 right answers. The vertical bar (|) separates the answers. This syntax specifies that 1 and 2 and 3, and so on up to 12, are all right answers. The response selects the multiplication table for the number the student enters. The same response-contingent instructions are executed for any of the numbers.

The variable x is defined at lesson level and can be used in any unit. The response-contingent instructions assign a value to this variable. The system variable RESPONSE is a string variable containing the student's response. The string in RESPONSE changes after every response.

The data type of x is integer, and a string cannot be assigned to an integer variable. The instruction ASSIGN x := RESPONSE does not work. The system function NUMBER converts a string to either an integer or a real number, depending on the use of the function. The arguments of functions are enclosed in parentheses. The function NUMBER has one argument; the argument is the string to be converted.

The instruction ASSIGN x : = NUMBER(RESPONSE) first evaluates the function and converts the characters in the variable to a number. Because the variable x is defined as integer, the number in integer form is assigned to x. After this instruction, x contains the number the student choose.

Then the DO revone instruction calls the unit revone to display one times table. How revone uses the variable x is explained when the FOR instruction is discussed.

Specifying Variables as Answers

The RIGHTV and WRONGV instructions can specify answers as the current contents of a variable. (Other uses of RIGHTV and WRONGV are discussed in Chapter 8.) Only the specification of the answer is different; the judging process is the same.

This lesson uses random numbers in the problems, so that students can repeat the practice set any number of times without seeing the same problems in the same order. You saw how the problems are displayed in the discussion of displaying variables. The numbers are generated in the unit practice.

The following example shows part of the unit practice.

```
DEFINE   c:INTEGER                   $$ Counter for problems.
SEED
FOR      c := 1,25                   $$ Do 25 problems.
.        ASSIGN   x:=RANDOMU(0,13)            $$ Assign values to x and y.
.        LOOP     x = old_x
.        .        ASSIGN   x := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_x := x
.        ASSIGN   y:=(RANDOMU(0,13))
.        LOOP     y = old_y
.        .        ASSIGN   y := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_y := y
.        DO       prac1                       $$ Display problem.
.        DO       return
ENDFOR
```

The FOR instruction repeats the instructions between FOR and the ENDFOR instruction 25 times for the 25 practice problems. The two LOOP instructions prevent x and y from having the same values for two successive problems. The FOR instruction and the LOOP instructions are explained in detail in the section about control logic.

The instructions ASSIGN x:=RANDOMU(0,13) and ASSIGN y:=RANDOMU(0,13) assign values to x and y. The values are generated by the RANDOMU system function.

The RANDOMU function generates random numbers. The two arguments define the range of possible values. The first argument is included in the range, but the second argument is not. Because 0 and 13 are integers, the function generates integers from 0 (included in the range) up to (but not including) 13.

Because these two instructions are inside the FOR structure, two random numbers are generated each time the unit practice calls the unit prac1. The SEED instruction (outside the FOR structure) seeds the random number generator so that each sequence of random numbers starts with a different number.

After generating the two random numbers and assigning them to x and y, the unit practice calls the unit prac1, which displays the problem and judges the response.

Here is the unit prac1.

```
UNIT      prac1               $$ Unit prac1 displays and judges one problem.
ASSIGN    z:=x*y              $$ Calculate right answer for these
PROMPT    "= "                $$ two numbers.
FCOLOR    BLUE
SIZE      3
AT        820                          $$ Display current problem.
WRITE     <<S,x>> x <<s,y>>
QUERY     *
RIGHTV    Z                            $$ The current value of Z is right.
.         AT        1120
.         SIZE      2
.         WRITE     You're right,
.         AT        1420
.         SIZE      3
.         WRITE     <<S,x>> x <<S,y>> = <<S,z>>
WRONG                                  $$ Anything else is a wrong answer.
.         AT        1120
.         SIZE      2
.         FCOLOR    RED
.         WRITE     NO, the right answer is
.         SIZE      3
.         AT        1420               $$ Display the right answer.
.         WRITE     <<S,x>> x <<S,y>> = <<S,z>>
.         JUDGE     STOP               $$ Stop judging, do not repeat query.
ENDQ
PROMPT    ">"
```

The first instruction is the name of the unit, as usual. The second instruction solves the problem. This instruction first evaluates the expression x*y by multiplying the two numbers currently in x and y. Then it assigns the result to the variable z. The variable z now contains the product of the two numbers generated in the unit practice.

Because the numbers are generated by the lesson, you cannot use the instruction RIGHT to specify the answer. The arguments to RIGHT must be text strings. The instruction RIGHTV accepts a variable name as its argument. When the lesson is executed, the student's response is converted to the same data type as the specified variable, then compared to the current value of the variable.

The lesson uses variables to display the problem, calculate the answer, and specify the answer. The student sees the current value of the variables, multiplies them, and types a response.

The variables x, y, and z are also displayed in the feedback for responses judged right and responses judged wrong. The two WRITE instructions in the response-judging block use the string format.

Modifying Response Judging

By default, the student must enter a response judged right before response judging stops. To have response judging end after the first response whether the response is right or wrong, the default process must be modified. The JUDGE instruction modifies the response-judging process. The arguments to JUDGE are keywords. The JUDGE instruction with the argument STOP stops judging. The next instruction executed is the instruction after the ENDQ.

After judging the response and displaying the feedback, unit prac1 ends. Control returns to the instruction following the DO prac1 instruction in the unit practice. The next instruction in practice is DO return, which displays the PRESS RETURN message and pauses.

Figure 5–7 shows the screen after a student has entered a response judged wrong.

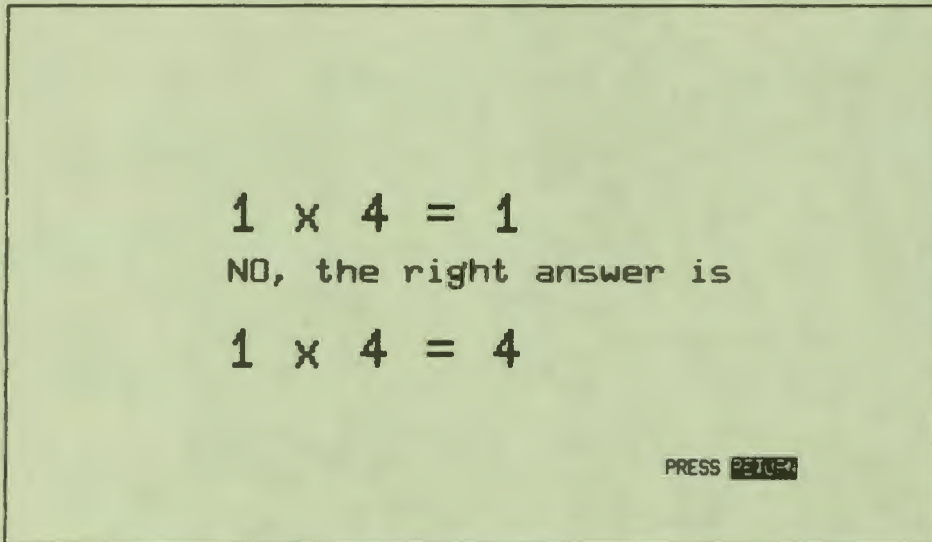


Figure 5–7
Problem Judged Wrong

SCORING AND GOALS

The lesson Multiply scores only the practice problems and keeps a separate score for each set of problems. There are two system variables for scoring. The variable SCORE contains the total score for all scored queries. The variable SCORES is an array, each element of which contains the score for one goal.

The unit practice controls both scoring and the goal structure of the lesson.

The unit practice follows.

```
UNIT     practice              $$ Displays 25 problems and scores them.
DEFINE   old_x,old_y:INTEGER
ASSIGN   old_x := 3            $$ Used to see if current x and y are
ASSIGN   old_y := 4            $$ same as x and y for last problem.
ERASE
BOX      0,0;0.999,0.999               $$ For manual illustrations.
FCOLOR   RED
SIZE     2

; The IF block is executed only the first time the student chooses
; to practice. The variable done_once is defined at lesson level and
; is assigned the value TRUE in this unit.
; IF     done_once = FALSE             $$ If this is the first time
.        do       instruc              $$ display instructions
ENDIF

SCORE    TRUE     $$ Begin scoring.

DEFINE   c:INTEGER             $$ Counter for problems.
SEED
FOR      c := 1,25                     $$ Do 25 problems.
.        ASSIGN   X:=RANDOMU(0,13)             $$ Assign values to x and y.
.        LOOP     x = old_x
.        .        ASSIGN   X := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_x := x
.        ASSIGN   Y:=(RANDOMU(0,13))
.        LOOP     y = old_y
.        .        ASSIGN   y := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_y := y
.        DO       prac1                        $$ Display problem.
.        DO       return
ENDFOR

SCORE    FALSE                 $$ Stop scoring.
DO       shoscore              $$ Display score for practice.
GOAL     GOAL+1 $$ Increment goal for next time.
```

At lesson level, the instruction SCORE FALSE turns scoring off. The students' responses at lesson level and in the unit review are not scored. The first time that the unit practice is executed, the unit instruc displays instructions and a practice problem. This problem is not scored either. After the unit instruc returns, the instruction SCORE TRUE turns scoring on. Then the FOR loop executes the unit prac1 25 times. Each of the 25 problems is scored. The SCORE FALSE instruction after the ENDFOR turns scoring off. When the unit returns to lesson level, scoring is off, and is not turned on again unless the student chooses to practice again.

The variable SCORE contains the total score for all responses. After the first set of problems, SCORE contains a number from 0 to 25. After the second set of problems, SCORE contains a number from 0 to 50, and so on, as long as the student chooses to practice.

One of the requirements for Multiply, however, is that students see their score for each set of practice problems. This requirement is met with the GOAL instruction, the GOAL variable, and the SCORES array. Making each set of practice problems a different goal stores the scores for each set in an element of the array SCORES. The index to this array is a goal number. The score for goal 1 is stored in SCORES(1); the score for goal 2 is stored in SCORES(2), and so on.

When a lesson begins, the current goal is goal 1 by default. The GOAL instruction begins a new goal and defines a new goal number. The system variable GOAL contains the current goal number. When a lesson begins, the value of this variable is 1. The argument to the GOAL instruction is a number that is assigned to the variable GOAL.

The last instruction in the unit practice is GOAL GOAL+1. The argument is an expression using the system variable GOAL. The first time the unit is executed, the value of the variable GOAL is 1. Evaluating the expression GOAL+1 reads the current value of the variable GOAL and adds one to it. Then the GOAL instruction stores the value of the expression in the system variable GOAL. After the student completes the first set of practice problems, the current goal is goal 2. The current value of the variable GOAL is 2.

If the student chooses to practice again, the score for the second set of problems is stored in SCORES(2). At the end of the unit practice, the system variable GOAL is incremented again. Now the current goal is goal 3. If the student practices again, the score is stored in SCORES(3).

The scores stored in the SCORES array are displayed in the unit shoscore.

CONTROL LOGIC

You have seen how the LOOP instruction and a response-judging block are used as control logic at lesson level. This section explains three more control logic structures: the FOR,ENDFOR structure; the IF,ELSE,ENDIF structure; and the TEST,VALUE,ENDTEST structure.

Both the LOOP structure and the FOR structure repeat a series of instructions. The two structures end the repetition differently.

The LOOP instruction tests the value of a Boolean variable or expression. If the expression is true, the instructions between LOOP and ENDLOOP are executed. Then the expression is tested again. If the expression is still true, the instructions are executed again, and so on. The repetition ends when the value of the Boolean expression is false. (This form of repetition is often called conditional iteration.)

The FOR instruction tests the value of a variable called the counter, and executes the instructions between FOR and ENDFOR if the value of the counter is less than or equal to an ending value. The ENDFOR instruction increments the counter; the FOR instruction tests the counter again, and so on, until the counter reaches the specified ending value. Then the FOR loop ends. The repetition ends after a specified number of times.

The IF instruction also tests a Boolean variable or expression. If the variable is true, the instructions following IF are executed once. The ELSE instruction is optional. When it is used, the instructions following ELSE are executed when the Boolean variable is false.

The TEST instruction tests a variable of any data type, and compares its value to numbers, strings, or other variables specified by any number of VALUE instructions. If the variable and the argument to one VALUE instruction match, the instructions following that VALUE are executed once. Deciding whether to execute instructions based on the value of a variable or deciding which of several sets of instructions to execute is called conditional execution.

The LOOP,ENDLOOP Structure

The unit practice generates random numbers for the practice problems. The unit uses LOOP instructions to assure that the same problem is not displayed twice in succession.

The complete unit is shown in the section on scoring. The instructions that affect the generation of the variables x and y are shown below.

```
DEFINE   old_x,old_y:INTEGER
ASSIGN   old_x := 3            $$ Used to see if current x and y are
ASSIGN   old_y := 4            $$ same as x and y for last problem.
; instructions omitted
DEFINE   c:INTEGER             $$ Counter for problems.
SEED
FOR      c := 1,25                       $$ Do 25 problems.
.        ASSIGN   X:=RANDOMU(0,13)             $$ Assign values to x and y.
.        LOOP     x = old_x
.        .        ASSIGN   X := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_x := x
.        ASSIGN   Y:=RANDOMU(0,13)
.        LOOP     y = old_y
.        .        ASSIGN   y := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_y := y
.        DO       prac1                        $$ Display problem.
.        DO       return
ENDFOR
```

The instruction DEFINE old_x,old_y:INTEGER defines two unit-level variables. Because these variables are defined at unit level, they can be used only in the unit. Every time the unit practice is executed, these variables are created. When the unit returns to lesson level, the variables no longer exist.

The instructions ASSIGN old_x := 3 and ASSIGN old_y := 4 assign initial values to the variables.

The instructions that assign a number to x before calling the unit prac1 to display and judge the problem are:

```
.        ASSIGN   X:=RANDOMU(0,13)             $$ Assign values to x and y.
.        LOOP     x = old_x
.        .        ASSIGN   X := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN   old_x := x
```

The instruction ASSIGN X:=RANDOMU(0,13) assigns a random number to x. The Boolean expression tested by the LOOP instruction is x = old_x. This expression is true when the numbers in the two variables are the same. When the expression is true, the instruction between LOOP and ENDLOOP is executed. This instruction assigns another random number to x. Then the LOOP instruction compares this number and old_x. If the numbers are still the same, another number is assigned to x, and so on.

If the numbers are different, the instruction in the loop is not executed; and the loop ends. In either case, x contains the number that is used for the next multiplication problem. The instruction ASSIGN old_x := x saves the current value of x in old_x. At this point, the two variables contain the same value.

The instruction DO prac1 later in the FOR loop displays and judges one problem using x. After the unit prac1 returns, the ENDFOR instruction returns control to the FOR instruction. The instruction ASSIGN x:=RANDOMU(0,13) assigns a new number to x. The variable old_x still contains the number from the last problem, so the expression x = old_x is true if the number displayed in the last problem and the number to be displayed in the next problem are the same.

At the beginning of the unit, the value 3 is assigned to old_x. The first time the student chooses to practice, the unit instruc calls the unit prac1 to display one problem. For this problem, x is 3. Initializing old_x as 3 assures that the first scored problem is not the same as the practice problem.

The FOR,ENDFOR Structure

The control logic for the unit review calls the unit revall if students choose to review all the times tables and the unit revone if students choose to review only one of the times tables. When students choose one, unit review assigns the number they enter to the variable x.

The unit revone uses a FOR instruction to assign the values 0 through 12 to y, to calculate the product of x and y, and to display the 13 lines of the times table.

Unit revone contains the following instructions.

```
; The unit revone displays one times table. It is called 12 times
; by unit revall or once by unit review.
UNIT     revone               $$ The value of x when revone is called is
AT       025                            $$ set by unit review or unit revall.
FCOLOR   BLUE
SIZE     2
FOR      y := 0,12                      $$ For 13 times, changing the value
.        ASSIGN   z := x*y              $$ of y each time, calculate answer
.        WRITC    <<T,x,2,0>> x <<T,y,2,0>> = <<T,z,3,0>>
ENDFOR                                  $$ and display it.
DO       return                         $$ Execute unit return.
```

The FOR instruction has three arguments. The syntax is:

```
FOR      counter := initial_value,ending_value
```

The variable y is the counter in the FOR instruction in unit revone. The initial value is 0. The ending value is 12.

The first time the FOR instruction is executed, the counter is assigned the initial value. In this example, the variable y has the value 0. Then the counter is compared to the ending value. If the counter is greater than the ending value, the instructions between FOR and ENDFOR are not executed. The first time, y is 0 and the ending value is 12. The counter is less than the ending value, so the instructions between FOR and ENDFOR are executed.

The instruction ASSIGN z := x * y calculates the product of x and 0, and assigns the value to z. The WRITC instruction displays the variables in tabular format.

The next instruction is ENDFOR. At this point, 1 is added to the counter. The variable y now contains the value 2. The FOR instruction is executed again and the counter is compared to the ending value. Two is less than 13, so the instructions are executed again. Because the WRITC instruction inserts a (RET) at the beginning of the line, the second line of the times table is displayed below the first line.

The iteration continues until y is assigned the value 13. This occurs after the multiplication for x times 12 is displayed. When the FOR instruction compares the value of the counter to the ending value, 13 is greater than 12. The instructions between FOR and ENDFOR are not executed again.

The instruction after ENDFOR is DO return. This unit displays the PRESS RETURN message and pauses until the student presses (RET). The instruction DO return is the last instruction in the unit revone. The unit revone returns to the unit review.

The unit revone is also called 12 times by the unit revall. The unit revall contains a FOR instruction that uses the variable x as the counter.

Here is the unit revall.

```
UNIT    revall
FOR     x:=1,12
        DO      revone
ENDFOR
```

The variable x is the counter, and its value is incremented by 1 at the ENDFOR instruction. Since the initial value is 1, the first value assigned to x is 1. The first time unit revone is called, the value of x is 1. The unit revone increments the value of y and displays the ones times table. Because unit revone is called from unit revall, it returns to unit revall. The FOR instruction increments x; revone displays the twos times table, and so on.

Figure 5–8 shows how the two variables change in the FOR loops in the two units.

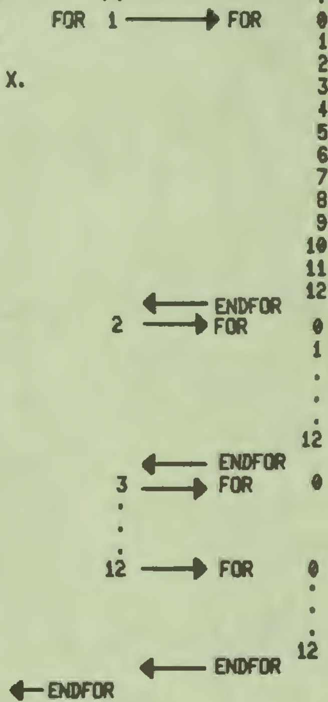


Figure 5–8
The FOR Instruction

The IF,ENDIF Structure

The IF instruction controls conditional execution of one or two sets of instructions. The syntax of an IF structure is:

```
IF      Boolean expression
.       instructions
ELSE
.       instructions
ENDIF
```

The ELSE instruction and the second set of instructions following it are optional. First the Boolean expression is tested. If the expression is true, the instructions following IF are executed. The instruction after ENDIF is executed next. If the expression is FALSE, the instructions following ELSE are executed, then the instruction following ENDIF. If there is no ELSE and the expression is false, no instructions in the IF structure are executed.

The following IF structure is in the unit practice.

```
IF        done_once = FALSE
.         DO        instruc
ENDIF
```

The unit instruc displays instructions for answering the multiplication, then displays one problem and judges the response to make sure the student understands.

The Boolean variable done_once is defined at lesson level. Its initial value is FALSE. The first time the student chooses to practice, the Boolean expression is true. The unit instruc is executed and changes the value of done_once to TRUE. If the student chooses to practice again, the expression is false; and the unit instruc is not executed.

The unit practice calls the unit shoscore after the student finishes the set of 25 problems. The unit displays the score for the current set of problems. If the student has practiced before, shoscore also displays the score for the previous set and compares the two scores.

The unit shoscore uses IF instructions to decide what to display. The unit shoscore follows.

```
UNIT      shoscore
FCOLOR    RED
SIZE      2
AT        505
TEST      SCORES(GOAL)        $$ Score for set of 25 problems.
VALUE     25                  $$ All right.
.         WRITE     VERY GOOD
                    YOU GOT THEM ALL RIGHT.
VALUE     20..24
.         WRITE     GOOD
                    YOUR SCORE IS <<S,INT(SCORES(GOAL))>>.
VALUE     0..19
.         WRITE     YOU SHOULD REVIEW BEFORE
                    YOU PRACTICE AGAIN.
                    YOUR SCORE IS <<S,INT(SCORES(GOAL))>>.
```

```
ENDTEST
AT      1305
IF      GOAL > 1                          $$ Practiced more than once.
.       WRITC   LAST TIME YOUR SCORE WAS <<s,int(scores(goal-1))>>
.       IF      SCORES(GOAL) <> SCORES(GOAL-1)
.       .       IF SCORES(GOAL) > SCORES(GOAL-1)
.       .       .       WRITC   YOU'RE DOING BETTER
.       .       ELSE
.       .       .       WRITC   YOU'VE SHOWN YOU CAN DO BETTER
                                TRY IT AGAIN.
.       .       ENDIF
.       ELSE
.       .       WRITE   TOO.
.       ENDIF
ENDIF
DO      return                            $$ Execute unit return.
```

The score for the set of problems just completed is displayed by the TEST structure at the beginning of the unit. This structure is explained in the next section. Figure 5–11 shows this display.

The IF instructions in the second part of this unit display the score for the previous set of problems, if there is one. Figure 5–9 shows this display.

```
GOOD
YOUR SCORE IS 22.



LAST TIME YOUR SCORE WAS 24
YOU'VE SHOWN YOU CAN DO BETTER
TRY IT AGAIN.
                                PRESS [illegible]
```

MR-S-2129-82

Figure 5–9
The Unit Shoscore

The IF instructions in the unit shoscore are nested: that is, one IF,ELSE,ENDIF structure is inside another. Figure 5–10 shows the three levels of nesting and summarizes conditions for executing the instructions at each level.

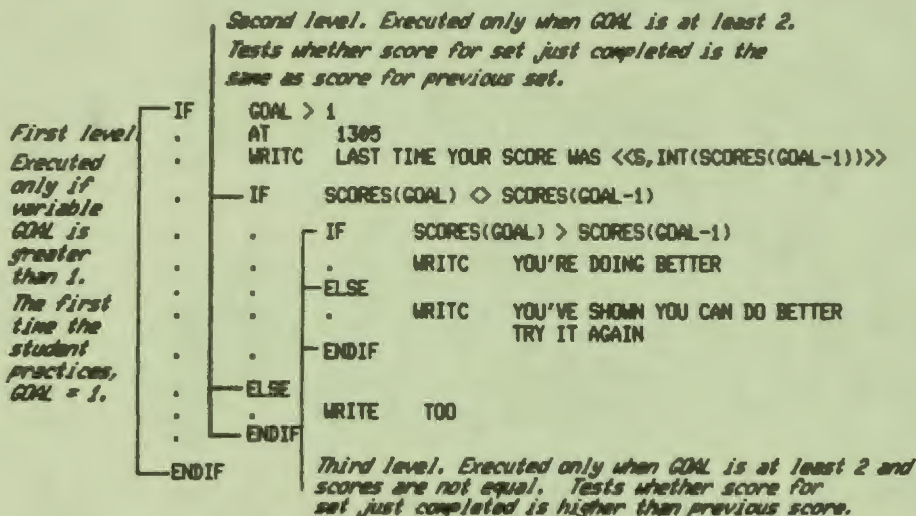


Figure 5–10
Nested IF Instructions

The Boolean expressions tested by all three IF instructions depend on the system variable GOAL and the SCORES array. How values are assigned to these variables is explained above. The system variable GOAL contains the goal number for the set of problems the student has just completed. The SCORES array is indexed by goal number, so the array element SCORES(GOAL) contains the score for the same set of problems.

The IF instruction at the first level tests the expression GOAL > 1. The operator > means greater than. When the student begins the lesson, the goal number is 1 by default. After the first set of problems, the variable GOAL contains the value 1; so this expression is FALSE. The goal number is not greater than 1. Because the expression is FALSE, none of the instructions between the first-level IF and ENDIF are executed.

After the second set of practice problems, the goal number is 2. Now the expression GOAL > 1 is true. The instructions between IF and ENDIF are executed. The WRITC instruction displays a line of text. The next instruction is the second-level IF instruction.

The expression tested by the second-level IF instruction compares two elements of the SCORES array. Because the current goal number is 2, the value of the variable GOAL is also 2. The array element SCORES(GOAL) contains the score for the current set of problems. The Boolean operator <> means not equal. The expression GOAL–1 used as the index to the other element of the SCORES array evaluates to 1 at this point. The element SCORES(GOAL–1) contains the score for the first set of problems. The expression is true when the two scores are not equal.

The second-level IF instruction has a matching ELSE instruction, so one of the two sets of instructions is always executed. When the two scores are equal — that is, when the expression tested by IF is false — the instructions following ELSE are executed. The instruction writes the one word TOO on the screen. The next instruction to be executed is the instruction following the ENDIF at the second level. The next instruction is the ENDIF instruction at the first level, so the conditional execution begun by the first-level IF instruction is also finished.

When the expression tested by the second-level IF instruction is true, the instructions between IF and the second-level ELSE instruction are executed. When the scores for the two sets of practice problems are not equal, the next step is to determine which is greater.

The expression tested by the third-level IF is true when the current score is greater than the previous score. In this case, the instructions between IF and ELSE are executed, and the student is told this score is better than the last one. The expression is false when the previous score is greater than the current score. In this case, the instructions between ELSE and ENDIF are executed, and the student is told the last score was better.

The TEST,VALUE,ENDTEST Structure

The TEST,VALUE,ENDTEST structure also performs conditional execution. The syntax of the structure is:

```
TEST     variable
VALUE    condition
.        instructions
VALUE    condition
.        instructions
OTHER
.        instructions
ENDTEST
```

The argument to TEST specifies a variable to be tested. The arguments to the VALUE instructions specify possible values of the variable. The variable is compared to the argument of the first VALUE instruction. If they match, the instructions after VALUE are executed; then the instruction after ENDTEST is executed. If they do not match, the variable is compared to the argument of the next VALUE instruction. The procedure continues until either the OTHER instruction or the ENDTEST instruction is reached. The OTHER instruction is optional. When OTHER is used, the instructions following it are executed when none of the VALUE instructions match.

The TEST structure from the unit shoscore is repeated below.

```
AT       505
TEST     SCORES(GOAL)
VALUE    25
.        WRITE    VERY GOOD
                  YOU GOT THEM ALL RIGHT.
VALUE    20..24
.        WRITE    GOOD
                  YOUR SCORE IS <<S,INT(SCORES(GOAL))>>
VALUE    0..19
.        WRITE    YOU SHOULD REVIEW BEFORE
                  YOU PRACTICE AGAIN.
                  YOUR SCORE IS <<S,INT(SCORES(GOAL))>>
ENDTEST
```

Figure 5–11 shows the display created by these instructions.

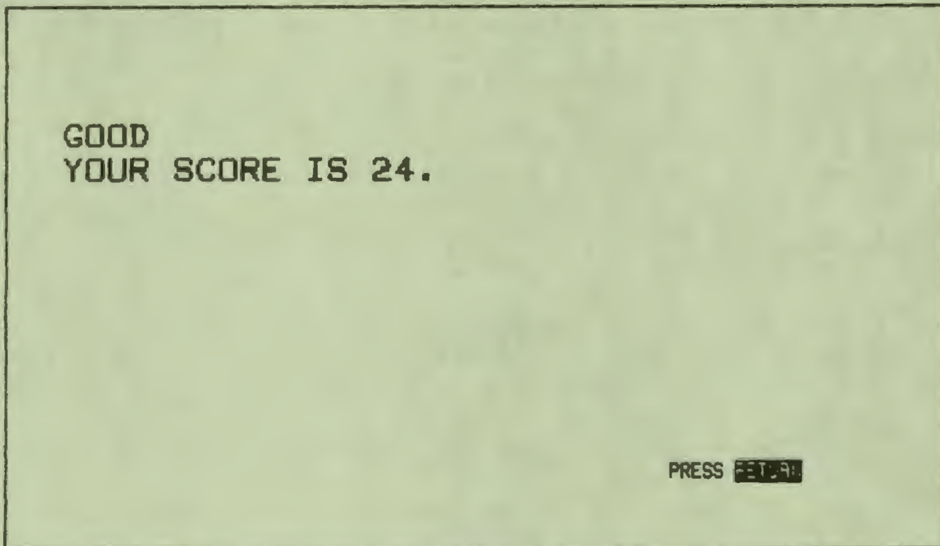


Figure 5–11
The TEST Instruction

The variable being tested is the array element SCORES(GOAL), which contains the score for the current set of problems. The score is a real number from 0 to 25.

The first VALUE instruction specifies the condition 25. The text block following this VALUE instruction is displayed if the student's score is 25. Since one argument of one of the VALUE instructions matched the variable, the instruction following ENDTEST is executed next.

If the student missed at least one problem, the score is compared to the argument of the second VALUE instruction. This argument specifies a range of numbers. If the score is 20, 21, 22, 23, or 24, it matches this range. If the score matches, the text block following this VALUE instruction is displayed.

If the score is in the range 0..19, it matches the argument of the third VALUE instruction; and third text block is displayed.

This TEST structure does not use the OTHER instruction. The OTHER instruction begins a set of instructions that are executed only when the variable being tested does not match the argument to any VALUE instruction. The arguments to the three VALUE instructions cover all possible scores. Because this lesson displays 25 problems and adds one to the score for each right response, the variable SCORES(GOAL) always contains a number from 0 through 25.

You could use the IF structure to make the same comparisons. In this case, the TEST structure is preferable for two reasons. First, there are three alternatives. You cannot nest them as the IF structures above were nested, and you would have to use three IF structures. Second, it is easier to specify a range with the TEST structure than with the IF structure.

Notice that the AT instruction that specifies the position for the feedback that is displayed is outside the TEST structure. In this way, one AT sets the current location for any of the three WRITE instructions. You could also use an AT instruction after each of the three VALUE instructions.

Combining Structures

You can combine control structures by nesting them inside each other. The section explaining the IF structure showed three levels of nested IF structures. Here is one final example showing a TEST structure nested in a LOOP structure.

The first time students choose to practice, the lesson explains what to do and gives them a chance to respond to one problem that is not scored. If the response is right, the lesson goes on to the 25 practice problems. If the response is wrong, the lesson displays the answer and asks the students to enter another response.

The unit instruc displays the instructions and the sample problem. The unit instruc follows.

```
UNIT      instruc
ASSIGN    X:=3        $$ assign values for practice problem.
ASSIGN    Y:=4
AT        503
WRITE     You will see 25 multiplication problems.
          Type the answer and press RETURN.
          You will be told if
          you got the answer right.

          Press RETURN again for the next problem.
          Press RETURN now to see a sample.

DO        return
LOOP      done_once = FALSE              $$ Display problem and loop
.         DO        prac1                $$ until student gets it right
.         SIZE      2
.         TEST      RESPONSEV
.         VALUE     12        $$ Practice problem answered correctly
.         .         AT        310
.         .         WRITE     GOOD — you seem to understand.
                              Now the real problems start.
.         .         DO        return
.         .         ASSIGN    done_once:=TRUE  $$ Assign new value
.         OTHER                          $$ to control variable.
.         .         AT        310
.         .         WRITE     Try again, now that you know
                              the right answer
.         .         DO        return
.         ENDTEST
ENDLOOP
```

Figure 5–12 shows the explanation displayed by the first part of this unit.

```
You will see 25 multiplication problems.
Type the answer and press RETURN.
You will be told if
you got the answer right.

Press RETURN again for the next problem.
Press RETURN now to see a sample.

                                   PRESS RETURN
```

MR-S-2132-82

Figure 5–12
The Unit Instruction

The variable done_once is defined at lesson level. An IF instruction in unit practice calls the unit instruc if the value of done_once is FALSE. The expression done_once = FALSE is tested by the LOOP instruction. If the expression is true, which it is the first time, the instructions between LOOP and ENDLOOP are executed. The instruction DO prac1 executes the unit prac1 to display the problem and judge the response.

The TEST instruction tests the system variable RESPONSEV. This system variable contains the student's response if the response matches the answer specified by the RIGHTV instruction in unit prac1.

In the default response-judging process, a response-judging block does not end until the student enters a response judged correct. Had the default process been used in unit prac1, the RESPONSEV variable would always contain the value 12 when the unit returns. In unit prac1, however, the JUDGE STOP instruction overrides the default response-judging process, so that judging stops after the first response whether it is judged right or wrong. Therefore, if the student's first response is wrong, RESPONSEV contains a value other than 12 when unit prac1 ends.

If the student's response in prac1 is correct, the instructions between VALUE and OTHER are executed. The student sees the display in Figure 5–13.

```
GOOD--you seem to understand.
    Now the real problems start.

        3 x 4 = 12
        You're right,
        3 x 4 = 12

                              PRESS RETURN
```

MR-S-2133-82

Figure 5-13
The Example Problem

After the display, the variable done_once is assigned the value TRUE. The lesson goes to the instruction following ENDTEST. This instruction is ENDLOOP, which transfers control back to the LOOP instruction.

The expression done_once = false is evaluated again. This time the expression is false because the value of done_once has changed. The instructions between LOOP and ENDLOOP are not executed. Since ENDLOOP is the last instruction in the unit, the unit returns to lesson level.

If the student's response to the sample problem is judged incorrect, the path is different. Since the system variable RESPONSEV contains a value other than 12, the instructions between OTHER and ENDTEST are executed, and the student sees a different display.

Then control is passed back to the LOOP instruction. This time, however, the variable done_once has not been changed, so the instructions between LOOP and ENDLOOP are executed again. The student has another chance to answer the sample problem.

ENDING THE LESSON

The last instruction in the lesson is ENDLESSON. This instruction follows the final instruction in the last unit, and has no effect on lesson execution. Actual execution ends at the last lesson-level instruction. The ENDLESSON instruction is a signal to the compiler that there are no more instructions in this lesson.

6

Editing, Compiling, and Linking a Lesson

6

Editing, Compiling, and Linking a Lesson

Three functions must be performed before a lesson can be executed: editing to produce a file containing source code in VAX DAL, compiling the source code to produce object code, and linking the object code and the run-time library for VAX DAL with the VAX/VMS linker.

Editors and the VAX/VMS linker are available to all VAX/VMS users and are documented in detail in the VAX/VMS document set. Programs that perform useful functions such as copying and printing files are also documented there.

The compiler for VAX DAL is explained in detail in this manual.

If you are an experienced VAX/VMS user, you may want to skip to the last section of this chapter which explains the compiler command line and options.

If you have never used a computer before, or if you are not familiar with VAX/VMS, you should read the *Introduction to VAX/VMS* before you start. The *VAX/VMS DCL Dictionary* also contains helpful information. The *EDT Editor Manual* explains how to use the editor EDT to edit source files.

Installations can customize the VAX/VMS software. Check with your system manager to see if your site has command procedures to simplify compiling and linking lessons.

PREPARING A LESSON

Once you have planned and designed your lesson, you must perform the following steps before the lesson can be executed.

1 Editing

The instructions that make up the lesson are entered into a source file using a computer program called an editor. You use an editor to type VAX DAL instructions into the computer file or files that make up the lesson.

Editors also make it easy for you to correct typing mistakes, insert and delete lines or whole units, and find lines before or after your current position in the file. The instructions in the file you edit are called source code. When you display the file on the terminal, you can read the instructions. The compiler can also read the instructions for the next step in the process.

2 Compiling

Compiling produces an intermediate file containing a version of your instructions called object code. While the compiler is reading your source file, it checks the syntax of your instructions. If there are syntax errors in your lesson, such as commas left out between arguments or IF instructions without matching ENDIF instructions, the compiler identifies and reports these errors.

3 Linking

Linking reads the object code generated by the compiler and produces a file that contains an executable image of the lesson.

When linking is complete, you have an executable lesson.

Before a lesson is ready for students to take, each of these steps is generally repeated several times. Compiling a lesson for the first time after it is edited almost always reveals syntax errors. The lesson is edited again to remove these. Executing the lesson for the first time may uncover other kinds of errors. Perhaps you have an ERASE instruction with arguments that erase too much or not enough. Perhaps you chose a text size that writes part of a display off the screen. The lesson must be edited, compiled, linked, and executed again.

Testing and Correcting a Lesson

Lessons require two distinct types of testing: technical and educational.

Prior to educational testing, a lesson is tested to ensure that it works according to your plan. While you are editing, compiling, linking, and executing your lesson during implementation, you may find three kinds of errors in the lesson: compiler errors, run-time errors, and logic errors.

When the lesson is finished and runs as planned, it needs to be tested for educational validity. This area of testing is beyond the scope of this manual.

Compiler Errors

The VAX DAL compiler tests the syntax of the instructions. For every error, the compiler displays a message on the terminal showing the instruction, its line number, and a description of the error.

The compiler reports all errors that it finds. Some errors are simply warnings. A lesson can be linked if it contains only warning errors.

Typical errors the compiler finds are: punctuation marks that are not part of the syntax for the instruction; instructions that have too few or too many arguments; missing ENDIF instructions; mismatched parentheses; and expressions with incompatible data types.

Here are some examples of error display on the screen, and explanations of the errors.

Example 1

- CODE

```
LESSON   BAD
DEFINE   X:READ
ASSIGN   X:=20
```

- ERROR DISPLAY

```
$ dal bad
   2    DEFINE  X:READ
%DAL-E-UNKDATATYPE, Unknown data type in DEFINE command
   3    ASSIGN  X:=20
%DAL-E-UNDEFVAR, Undefined variable
%DAL-E-INVASGNDEST, Invalid destination for the ASSIGN command
%DAL-W-NOENDLESSON, No ENDLESSON command found

%DAL-E-ENDCMPERR, Compilation complete with errors
$
```

MR-S-3965-85

- EXPLANATION

  The error in the code is a simple typing mistake: the data type in the DEFINE instruction should be REAL. The first error message identifies the error. This error also produces the two error messages for the second line of code. The compiler cannot define a variable without a data type, so the variable x is not defined. A value cannot be assigned unless the variable is defined. These errors mean that the lesson cannot be linked and executed.

  A single error may produce a number of error messages. Because the data type READ does not exist, the variable X is not defined. Every reference to the variable X produces an error message. In a lesson with several hundred lines of code, a simple error like this one can produce twenty or thirty error messages. Errors such as forgetting an ENDIF or an ENDFOR can also produce many error messages because the compiler cannot close the structure.

  The last error message is produced because there is no ENDLESSON instruction. As the error message indicates, this is a warning message only. If this were the only error, the lesson could be linked and executed.

Example 2

- CODE

```
LESSON    BAD
DEFINE    X:REAL
FOR       X:=1,23
.         AT        X*100
WRITE     LINE <<S,X>>
ENDFOR
ENDLESSON
```

- ERROR DISPLAY

```
$ DAL BAD
   4            AT      X*100
%DAL-E-INVCROSSTYP, The CROSS coordinate specification must be of type integer
%DAL-E-INVCHRCOORD, Invalid character in coordinate specification
   5    WRITE   LINE <<S,X>>
%DAL-W-ILLDOTIND, Illegal dot indentation
   6    ENDFOR
%DAL-W-ILLDOTIND, Illegal dot indentation

%DAL-E-ENDCMPERR, Compilation complete with errors
$
```

MR-S-3939-85

- EXPLANATION

  These instructions are intended to write the word LINE and the value in the variable X on rows 1 through 23. The author expected the value of X to be 1 the first time FOR is executed. Multiplying by 100 gives 100 as the argument to AT. The second time, X is 2 and the argument to AT is 200. If you look only at the FOR instruction, this should work. All the numbers are integers.

  The problem is the data type of the variable X. Because X is defined as REAL, it contains 1.0, 2.0, and so on through the FOR loop. The automatic data type conversion performed by evaluating the expression X*100 produces real numbers. When real numbers are used as arguments to AT, they specify normalized coordinates.

  The compiler recognizes that this instruction is wrong, and assumes that row-and-column addresses are wanted. The two possible corrections are: change the DEFINE instruction so that X is defined as INTEGER, or use the system function INT to change the real number to an integer only for the AT instruction. In the second case, the AT instruction would be AT INT(X) * 100.

  The last error message is a dot indentation error. This line is inside the FOR,ENDFOR structure and should be indented. Like the missing ENDLESSON instruction in the previous example, this error does not stop compilation. The lesson can be linked and executed if there are no other errors.

Some errors in your code are obvious as soon as the compiler reports them. Others may not be easy to identify. Edit the lesson, correcting the obvious errors, and compile again. This step may eliminate some of the unidentified errors as well.

Run-time Errors

Run-time errors appear when you execute the lesson. A lesson executes up to a run-time error, and then stops and displays an error message.

Run-time errors often occur because of the current value of a variable. For example, you may be using a variable to store the position of the prompt character displayed by the QUERY instruction. When the lesson is executed, this variable contains different numbers at different times. A run-time error occurs if the prompt is off the screen.

Some run-time errors such as the example above are peculiar to lessons written in VAX DAL. Others are reported by VAX/VMS libraries and functions that are called by VAX DAL. Error messages produced by VAX/VMS libraries are listed in the VAX/VMS documentation.

An example of the display of run-time error messages follows with an explanation of what you should look for in the message and in your program.

- CODE

```
LESSON    BAD
AT        1010
SIZE      4
WRITE     This text is
          intended to
          make the prompt
          character go off
          the screen and
          display a
          run-time error
QUERY
RIGHT
ENDQ
ENDLESSON
```

- ERROR DISPLAY

```
%DAL-E-CUROUTBOU, Cursor out of bounds. Text will not be visible in INPUT/QUERY
command
$
```

MR-S-3940-85

- EXPLANATION

  When you execute this lesson, the text is displayed briefly. When the lesson attempts to display the prompt character, the address is checked. This address is off the screen. The lesson ends, erases the screen, and displays the error message.

  Given the text size and the address selected with AT, there are too many lines of text. The error can be corrected by selecting a different text size or a different starting address. It may be necessary to redesign the screen display.

Logic Errors

After compiler errors and run-time errors are identified and corrected, the lesson still may not do what you intended because of logic errors. Legitimate instructions that do something different from what you intended produce logic errors.

The following instruction is probably a logic error.

```
ERASE     510,1240
```

This instruction is legitimate. It erases one dot at fine coordinates 510,1240. However, it seems unlikely that the author intended to erase one dot that is not even visible on the screen.

A much more likely instruction is:

```
ERASE     510;1240
```

The change in punctuation means the arguments define the area to be erased in row and column coordinates. Now the instruction erases a rectangle beginning at row 5, column 10 and ending at row 12, column 40.

Other simple logic errors include forgetting a PAUSE instruction, changing text size or color and forgetting to change it back, and putting instructions in the wrong order. Generally, simple logic errors are easy to see when you execute the lesson, and easy to find and correct.

Logic errors can be more complicated and more difficult to find and correct. Usually, the behavior of the lesson indicates what part of the lesson is wrong. The problem can often be traced to a single unit. Inserting extra WRITE instructions to display the contents of variables at different times can be helpful. Using extra PAUSE instructions, especially in loops, can also help isolate an error.

THE VAX DAL COMPILER

The DAL compiler is executed by the command DAL. The name of the source file is a parameter for the command. Command switches modify the operation of the compiler.

A lesson can be edited in more than one source file. The compiler can combine several source files into one object file. If files are combined, there can be only one LESSON instruction. The LESSON instruction must be at the beginning of the first file. Lesson level ends at the first unit instruction in the combined files.

Compiler Switches

You have several options for compiler operation. Select an option by specifying one or more of the switches listed below. Switches listed in pairs are mutually exclusive; do not specify both of the switches in a pair.

- UNITS/NOUNITS

  The /UNITS switch displays the name of each unit as the compiler encounters it. The /NOUNITS switch turns off the display. The default is /NOUNITS.

- LIST/NOLIST

  The /LIST switch generates a file containing the source code during execution of the compiler. The file is not spooled to the printer automatically, but can be printed or displayed. The /NOLIST switch does not generate the file. The default is /NOLIST.

  If the /LIST switch is specified, a file name and extension can be specified. The default file name is the name of the source file. The default extension is .LIS. The format is /LIST = newname.ext.

- OBJ/NOOBJ

  The /OBJ switch produces a filename.OBJ file containing the compiled code. The /NOOBJ switch has no effect on the execution of the compiler or the operation of the other switches; however, no filename.OBJ file is created. The default is /OBJ.

  When the /OBJ switch is specified, the name of the .OBJ file can also be specified. The format is /OBJ = newname. The default name is the name of the source file.

- OLD_VERSION

  The OLD_VERSION switch compiles the lesson using the VAX DAL V1.1 compiler, rather than the V1.5 compiler. The /OLD_VERSION switch enables you to continue to modify and use lessons written in VAX DAL V1.1. If you use the /OLD_VERSION switch you must link the lesson to the V1.1 run-time library.

- /PUBLISH

  The /PUBLISH switch places a software lock on a lesson. Students can access lesson compiled with the /PUBLISH switch only through the Courseware Authoring System (C.A.S.) Delivery System. Attempts to run a lesson compiled with the /PUBLISH switch result in run-time error messages indicating that the lesson cannot be run. The /PUBLISH switch prevents students from setting their default directories to a directory containing the lesson's executable image and running the lesson without being detected or scored.

Compiler Examples

The following examples show how to call the compiler with different options. Switches must be entered on the same line with the command. You can enter the source file name either before or after the switches. If you do not enter the file name, you are prompted for it. The default extension for files is .DAL. You do not need to enter this extension.

The following example shows the command and the prompt with no switches. When the system prompt is displayed again, the compiler is finished.

```
$ DAL lesson
$
```

The following example shows the command with the source file name and the /UNITS switch entered on the same line. The name of each unit is displayed as the compiler encounters it. If the file is long, this option is useful as an indication that the compiler is working.

```
$ DAL/UNITS plurals
  INTRO
  QUERY1
  QUERY2
  QUERY3
  SHOWTEXT
$
```

The following example combines three source files into one object file and assigns a new name to the object file. The first file contains the LESSON instruction and all lesson-level instructions. The second and third files contain units. The last instruction in the third file is the ENDLESSON instruction.

```
$ DAL begin,middle,end/OBJ = lesson
$
```

THE VAX/VMS LINKER

The *VAX/VMS Linker Reference Manual* contains complete documentation for the linker. This section shows the command line for linking object files with the DAL run-time library to produce an executable image.

The command line is:

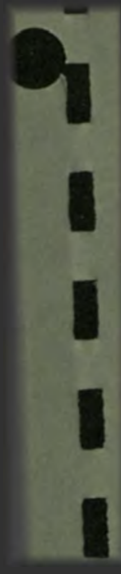
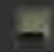
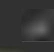
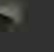
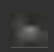
```
$ LINK filename{,filename...}
```

Where:

> *filename* is the name of the .OBJ file. Several file names can be specified and separated by commas.

7

Terminal Management

7

Terminal Management

A VAX DAL lesson can be delivered on any one of eight different DIGITAL terminal models. Some technical differences between terminal models can have an effect on lesson execution. Lessons developed and tested on one terminal model may behave differently on another.

This chapter suggests how to design lessons that can be displayed at terminals with different characteristics. The chapter begins with a description of the different characteristics of GIGI, VT125, VT240, VT241, DECmate II and III, Rainbow, and Professional terminals. The chapter also explains how to set various terminal characteristics on your terminal.

Topics discussed in this chapter include:

- How to program around hardware differences so that a lesson can be delivered at a variety of terminals
- How to adjust terminal characteristics, such as response echoing and response typeahead, on the terminal model you use
- How to set up the terminal so that a lesson can make use of special function keys
- How to save and restore terminal states

Much of the information contained in this chapter is also relevant to the discussion in Chapter 12 of the VAX DAL color management system. Authors should study both this chapter and Chapter 12 if they plan full color lessons to be displayed at a variety of terminals.

DIFFERENCES BETWEEN TERMINAL MODELS

A lesson developed and tested on one terminal model and executed on another can, among other things, place text or graphics off the screen or fail to find colors you request with FCOLOR or BCOLOR instructions. The technical differences that affect lesson execution can be grouped together into three main categories: software support requirements, screen characteristics, and color capabilities.

Software Support Requirements

A VAX DAL lesson requires different software support from different types of terminals. On many terminal models, a DAL lesson needs nothing more than its own built-in interconnection with the VAX/VMS terminal management facilities and ReGIS graphics facilities. On other terminal models, DAL requires auxiliary software. Users of any terminal model should include the VAX/VMS "SET TERMINAL/INQUIRE" command in their login command file. This command causes the VAX/VMS operating system to ask the user's terminal for its terminal characteristics. In this way, the VAX/VMS terminal management facilities are assured of having the correct attributes for the terminal in use. Table 7–1 lists the software support DAL requires of the different terminal models.

Table 7–1: VAX DAL Software Support Requirements

Terminal Model	Support Required
GIGI (VK100) VT125 VT240 or VT241	GIGI, VT125, VT240, and VT241 users should include a VAX/VMS SET TERMINAL/INQUIRE command in their command files. This ensures that VAX/VMS terminal management facilities use the correct attributes for the terminal
DECmate II & III	To use DECmate II or III terminals with DAL, the DECmate must be configured with the hardware graphics option and the DECmate Graphics Terminal Emulator V2.0 software. The graphics terminal emulator capabilities in DECmate WPS V2.0 and the DECmate Graphics Terminal Emulator V1.0 software are not supported. The DECmate Graphics Terminal Emulator (GTE) set-up option for terminal ID should be set to VK100 MODE–VT125 ID. Once this has been set correctly, the user's login command file should include a VAX/VMS SET TERMINAL/INQUIRE command.

Table 7–1: VAX DAL Software Support Requirements (Cont.)

Terminal Model	Support Required
Rainbow	To use a Rainbow with DAL, the Rainbow must be configured with three supporting software products: the hardware graphics option, Rainbow Poly-TRM, and Rainbow ReGIS. Poly-ReGIS is not supported. Once the Rainbow has been configured correctly, issue the following VAX/VMS command: SET TERMINAL/DEVICE = VT200/REGIS.
Professional	To use a Professional with DAL, the Professional must be configured with the extended bitmap graphics hardware option and PRO/Communications software, V2.0 or later. If PRO/Communications V2.0 is used, PRO/Communications should be set such that the terminal ID is VT125. (DAL does not support PRO/Communications V2.0 when the terminal ID is set to PRO mode.) Once this is correctly set, the user's login command file should include a VAX/VMS SET TERMINAL/INQUIRE command. This will ensure that VAX/VMS terminal management facilities have the correct attributes for a Professional terminal.

Screen Characteristics

Some terminal models have narrower screens than others. This means that displays that make use of the edges of the screen are not always transportable to other terminals.

Screen width is expressed in fine address units. A single fine address unit is the smallest unit on the screen that can be illuminated. Table 7–2 lists the different terminal models and their screen widths.

Table 7–2: Screen Widths

Terminal model	Number of units
GIGI (VK100) VT125 DECmate II & III Professional	767
VT240 VT241 Rainbow	799

Authors can make their lessons transportable between terminals with different screen widths if they use relative addressing or design their lessons to write only to screen addresses within the boundaries of the narrower screen.

Color Capabilites

For VAX DAL, the most significant differences between terminal models are in the area of color capabilities. For example, GIGI terminals have access to eight different colors, and all eight colors can appear on the terminal screen simultaneously. Rainbow terminals have access to 4096 different colors, but only four colors can be displayed on the terminal screen simultaneously

Authors must be aware of the color capabilities of both the terminals on which they develop their lessons and the terminals on which their lessons execute. A lesson fails to execute as expected if it uses more colors simultaneously than the terminal can support. A lesson also fails execute as expected if it requests colors that the terminal cannot provide.

Terminals use *color specifications* to generate colors. Color specifications are codes that define how the colors are generated. A terminal can only generate a color if it has the specification for that color.

If you are developing a lesson that is to be executed from several different types of terminals, plan to use only colors that all the terminals have specifications for. Also, different terminals support different numbers of colors on the screen simultaneously. Some terminals can support eight colors in a single display. Other can support only four. Plan to use only that number of colors in displays that can be supported by all of the terminals in use.

The table below contains the following information for each terminal:

- The number of color specifications the terminal has
- The number of colors that can appear on the screen simultaneously (including background color)
- The method used to store (and access) the color specifications

Table 7–3: Terminal Color Capabilities

GIGI (VK100)
- 8 colors available
- 8 colors supported simultaneously
- accessed by color-number (0–7) only

VT125
- 64 colors available
- 4 colors supported simultaneously
- accessed by HLS specification only

VT240 or VT241 (with ReGIS)
- 64 colors available
- 4 colors supported simultaneously
- accessed by HLS specification only

DECmate II
- 16 colors available
- 4 colors supported simultaneously
- accessed by color-number (0–15), or by HLS specification

DECmate III
- 64 colors available
- 4 colors supported simultaneously
- accessed by HLS specification only

Rainbow
- 4096 colors available
- 4 colors supported simultaneously
- accessed by HLS specification only

Professional Series (with PRO/Communication V2.0)
- 64 colors available
- 8 colors supported simultaneously
- accessed by HLS specification only

Chapter 12 in this guide contains a detailed discussion of the VAX DAL color management system and the DAL instructions that access and use color specifications. The Programmer's Reference Guide for each terminal lists the color specifications available with the terminal. Authors should refer to these two sources for more information about terminal color capabilities.

THE VAX DAL TERMINAL MANAGEMENT INSTRUCTIONS

VAX DAL is equipped with instructions that can tailor terminal characteristics to suit a lesson. These instructions do the following:

- Manage the way student responses are displayed on the screen
- Modify the color management capacities of the terminal
- Enable student use of special function keys
- Save and restore sets of terminal characteristics

Response Display Modifiers

Two instructions, SET ECHO and SET TYPEAHEAD, modify the way student responses are handled by the terminal and displayed on the terminal screen. SET ECHO OFF instructs the terminal not to display a student's response as it is typed. SET ECHO ON turns on the echoing of student responses on the screen. Responses are echoed on the screen by default.

When SET TYPEAHEAD ON is in effect, students can type responses before the response prompt appears. The terminal uses a typeahead buffer to store anything a student types before the prompt. If SET TYPEAHEAD OFF is in effect, each time a response prompt appears the terminal discards the contents of the typeahead buffer, and therefore discards anything the student typed before the prompt was displayed. If SET TYPEAHEAD ON is in effect, DAL treats the contents of the typeahead buffer as part of the student's response. When a response prompt appears, the contents of the typeahead buffer are displayed after the prompt. Responses can be typed before the prompt by default.

DAL Color System Modifiers

As mentioned above, different terminal models have widely different color capabilities. Two instructions, SET MAXCOLORS and SET HLS, modify the DAL color system by limiting the size of DAL's internal color table, and by establishing whether the terminal and supporting software in use require the HLS method for specifying colors.

The SET MAXCOLORS Instruction

SET MAXCOLORS modifies the size of the DAL color table. DAL uses the color table to identify the colors currently available for use. (See Chapter 12 in this guide for a complete discussion of the DAL color table.) If the terminal can support only four colors on the screen at the same time, the DAL color table contains four slots. If the terminal can support 16 colors on the screen at the same time, the DAL color table contains 16 slots. A lesson that uses more than four colors on the screen simultaneously fails to execute properly on a terminal that supports only four colors. After the color table of the four-color terminal fills up, subsequent FCOLOR instructions are ignored. A four-color terminal supports a maximum of four different colors on the screen at one time, including background color.

Authors developing lessons for terminals with different maximum color table sizes have two options. They can create several different versions of a lesson, with each version tailored to the color table of the terminal the lesson uses. Or, they can use the SET MAXCOLORS instruction to create a lesson that executes successfully on all of the different terminal models. The SET MAXCOLORS instruction establishes a size for the DAL color table. With SET MAXCOLORS, authors can limit the lesson color table to a size that all the terminals in use can support. The syntax of the SET MAXCOLORS instruction is:

SET (TAB)/(SP) MAXCOLORS, value

Where:

> *value* is a number from 1 to 64 that specifies the number of slots in the DAL color table.

Note that the color table cannot consist of more than 64 slots. See Chapter 12 for further discussion of the SET MAXCOLORS instruction.

The SET HLS Instruction

The SET HLS instruction, though used less frequently than the SET MAXCOLORS instruction, also modifies the DAL color system. Whenever a lesson makes use of a terminal or software utility that references color specifications by color number rather than HLS specification, SET HLS OFF can be used to turn off the use of HLS specifications. Authors rarely need to use this instruction: DAL automatically sets the terminal to the proper HLS setting at lesson startup.

Instructions That Enable Special Function Keys

Three instructions, SET FKEY, SET KEYPAD, and SET DELETE, enable lessons to detect student-pressed keys and treat those keys as student responses.

The SET FKEY Instruction

Function keys, the F-keys across the top of the keyboard of your terminal, generate escape sequences when pressed. If SET FKEY ON is in effect, DAL intercepts the escape sequence generated by a special function key and converts the sequence into an ASCII string. DAL then loads the ASCII string into the RESPONSE system variable where it can be treated just as if the student had actually typed in the string. Authors can test for key-generated ASCII strings in judging statements such as RIGHT and WRONG, or in control structures such as TEST/VALUE and IF structures. Key-generated ASCII strings can also be used in WHEN STRING statements to cause specific units to execute whenever students press specific function keys.

This special function key feature cannot be used with Professional series terminals.

The syntax of the SET FKEY instruction is:

SET (TAB)/(SP) FKEY, argument

Where *argument* can be:

> *ON*, which turns the special function key feature on. When SET FKEY specifies ON, ASCII strings are generated and trapped in the RESPONSE variable when students press function keys. Pressing a function key does not terminate the students' responses. Students still must use the RETURN key, or the current response delimiter, to terminate their responses.

OFF, which turns the special function key feature off. When SET FKEY specifies OFF, DAL intercepts and discards the escape sequences generated by special function keys. The function key feature is off by default.

TERMINATE, which turns the special function key feature on in the same way that SET FKEY, ON does. When SET FKEY specifies TERMINATE, a function key generates an ASCII string, and terminates the student's response. Students need not enter a response delimiter after they press the special function key.

When SET FKEY specifies ON or TERMINATE, the following ASCII strings are produced and entered in the RESPONSE variable when special function keys are pressed.

Table 7–4: ASCII Strings Generated by Special Function Keys

KEY(S)	STRING	KEY(S)	STRING
F07	[F07_KEY]	F08	[F08_KEY]
F09	[F09_KEY]	F10	[F10_KEY]
F11	[F11_KEY]	F12	[F12_KEY]
F13	[F13_KEY]	F14	[F14_KEY]
HELP	[F15_KEY]	DO	[F16_KEY]
F17	[F17_KEY]	F18	[F18_KEY]
F19	[F19_KEY]	F20	[F20_KEY]
FIND, E01	[E01_KEY]	INS, E02	[E02_KEY]
REM, E03	[E03_KEY]	SEL, E04	[E04_KEY]
PRV, E05	[E05_KEY]	NXT, E06	[E06_KEY]
←	[LFA_KEY]	→	[RTA_KEY]
↓	[DNA_KEY]	↑	[UPA_KEY]
PF1	[PF1_KEY]	PF2	[PF2_KEY]
PF3	[PF3_KEY]	PF4	[PF4_KEY]

Note that special function keys 1 through 6 cannot be used by a DAL lesson. These function keys are reserved for VAX/VMS operating system use. As a general rule, function keys 6 through 10 also should not be used. Depending on the system requirements at your location, function keys 6 through 10 may be used by the operating system.

The SET KEYPAD Instruction

Like SET FKEY, the SET KEYPAD instruction causes keys pressed by students to generate ASCII strings that can be treated as the students' responses. SET KEYPAD affects the keys on the numeric keypad of the terminal. ASCII strings are generated only if the keypad is in application mode (see below) *and* the SET FKEY instruction is set to ON. The syntax of the SET KEYPAD instruction is:

SET (TAB)/(SP) KEYPAD, argument

Where *argument* can be:

> *APPLICATION*, which sets the keypad to application mode. If SET FKEY specifies ON or TERMINATE, and SET KEYPAD specifies APPLICATION, keys on the keypad generate ASCII strings that are entered in the RESPONSE variable and can be treated as a student's response. When SET FKEY is ON, numeric keypad keys do not terminate the student's response. If SET FKEY TERMINATE is in effect, keys on the keypad enter and terminate the student's response.
>
> *NUMERIC*, which sets the keypad to numeric mode — keypad keys function normally generating numbers, ".", ",", "-", and RETURN. The keypad is in numeric mode by default.

Table 7–5: ASCII Strings Generated by Keypad Keys

KEY(S)	STRING	KEY(S)	STRING
0	[KP0_KEY]	1	[KP1_KEY]
2	[KP2_KEY]	3	[KP3_KEY]
4	[KP4_KEY]	5	[KP5_KEY]
6	[KP6_KEY]	7	[KP7_KEY]
8	[KP8_KEY]	9	[KP9_KEY]
,	[KPC_KEY]	-	[KPM_KEY]
.	[KPP_KEY]	ENTER	[KPE_KEY]

The SET DELETE Instruction

One final instruction modifies key usage at the terminal: the SET DELETE instruction. SET DELETE enables or disables use of the DELETE key. SET DELETE also makes it possible to detect student use of the DELETE key and record the use as an ASCII string in the RESPONSE system variable. The syntax of SET DELETE is:

SET (TAB)/(SP) DELETE, argument

Where *argument* can be:

ON, which is the default. When SET DELETE specifies ON, the DELETE key functions normally: pressing the key erases the character at the immediate left of the cursor. In this mode it is not possible to detect student use of the DELETE key. DELETE is not reported in the system variable KEYPRESSED and is not counted as a response character by the QLENGTH system variable.

OFF, which modifies the functioning of the DELETE key. When SET DELETE specifies OFF, pressing the DELETE key does not delete the character to the left of the cursor. Instead, the ASCII string [DEL_KEY] is inserted in the RESPONSE system variable and KEYPRESSED is set with the value 127. When SET DELETE is OFF, KEYPRESSED counts the DELETE key as one character in the student's response. The SET FKEY instruction does not need to be set to ON for the SET DELETE, OFF instruction to work.

Like ASCII strings generated by special function and keypad keys, the ASCII string generated by the DELETE key can be used in response judging or in control structures just as if the student had actually typed in the string.

All special function keys and keypad keys generate strings that have the character sequence "_KEY]" in common. Consequently, authors can use a single WHEN STRING statement to trap entry of any special function key used by the student. Or, the author can create individual WHEN STRING conditions for specific keys.

ASCII strings generated by special function keys, keypad keys, or the DELETE key are not echoed on the terminal screen. However, the string can be written to the screen from the RESPONSE variable after one of the keys is pressed.

Lesson Menu_Driver, listed in full in Appendix F, shows one application of special function keys in a lesson. Function keys in this instance are used in support of a menu driver.

Instructions That Save and Restore Terminal States

VAX DAL instructions make it possible to save sets of terminal characteristics, and later restore the characteristics to the terminal. Using SAVE and RESTORE instructions, authors can move a lesson back and forth between complete terminal states and complete sets of display attributes, without having to reset each attribute of the different terminal states with each transition.

The SAVE Instruction

Authors can create up to 10 different sets of terminal characteristics with the SAVE instruction. Each set can contain up to 17 different terminal characteristics.

The syntax of the SAVE instruction is:

SAVE (TAB)/(SP) set_number, item_to_save{,item_to_save ... }

Where:

set_number is a number from 1 to 10 that identifies a set of terminal characteristics.

item_to_save is a keyword that specifies a particular terminal characteristic to save. Up to 17 keywords, separated by commas, can be specified in each SAVE instruction. The keywords and the characteristics they save are listed in the table below.

Table 7–6: Terminal Characteristics Saved by the SAVE Instruction

Keyword	Saved
BCOLOR	Current BCOLOR setting
CHARSET	Name of the current character set
FCOLOR	Current FCOLOR setting
GORIGIN	GORIGIN X and Y coordinates
GWHERE	GWHEREX and GWHEREY, which are the X and Y coordinates of the last points expressed in the current graph scale
ITALICS	Current degree of italics
MODE	Current writing mode
PATTERN	Current writing pattern
PROMPT	Current prompt character
RORIGIN	Current RORIGIN X and Y coordinates
ROTATE	Current angle of rotation
RWHERE	X and Y coordinates of the point last drawn relative to the current RORIGIN
RSIZE	Current X and Y size coefficients (RSIZEX and RSIZEY)

Table 7–6: Terminal Characteristics Saved by the SAVE Instruction (Cont.)

Keyword	Saved
SIZE	Current text size
SREF	Coordinates and character pattern specified by the last SREF instruction
TROTATE	Current angle of text rotation
WHERE	Coordinates of the last point displayed (the values of the WHERE, WHEREX, and WHEREY system variables)

The RESTORE Instruction

Authors can restore a saved set of terminal characteristics to the terminal with a RESTORE instruction. RESTORE takes as its argument the set number of the set of characteristics it is to install. The syntax of the RESTORE instruction is:

RESTORE (TAB)/(SP) set_number

Where:

set_number is the number of a set of previously saved terminal characteristics.

The lesson below demonstrates the use of both the SAVE instruction and the RESTORE instruction.

```
LESSON   save_restore
;
;        This program shows the effect of saving and
;        restoring three sets of DAL terminal characteristics.
;
;        Set up the initial characteristics:
;
BCOLOR   blue
FCOLOR   white
SREF     500,400
PATTERN  dash
ITALICS  20
RORIGIN  600,100
;
;        Save these characteristics as three different terminal states.
;
SAVE     1,bcolor,fcolor
SAVE     2,sref,pattern
SAVE     3,italics,rorigin
```

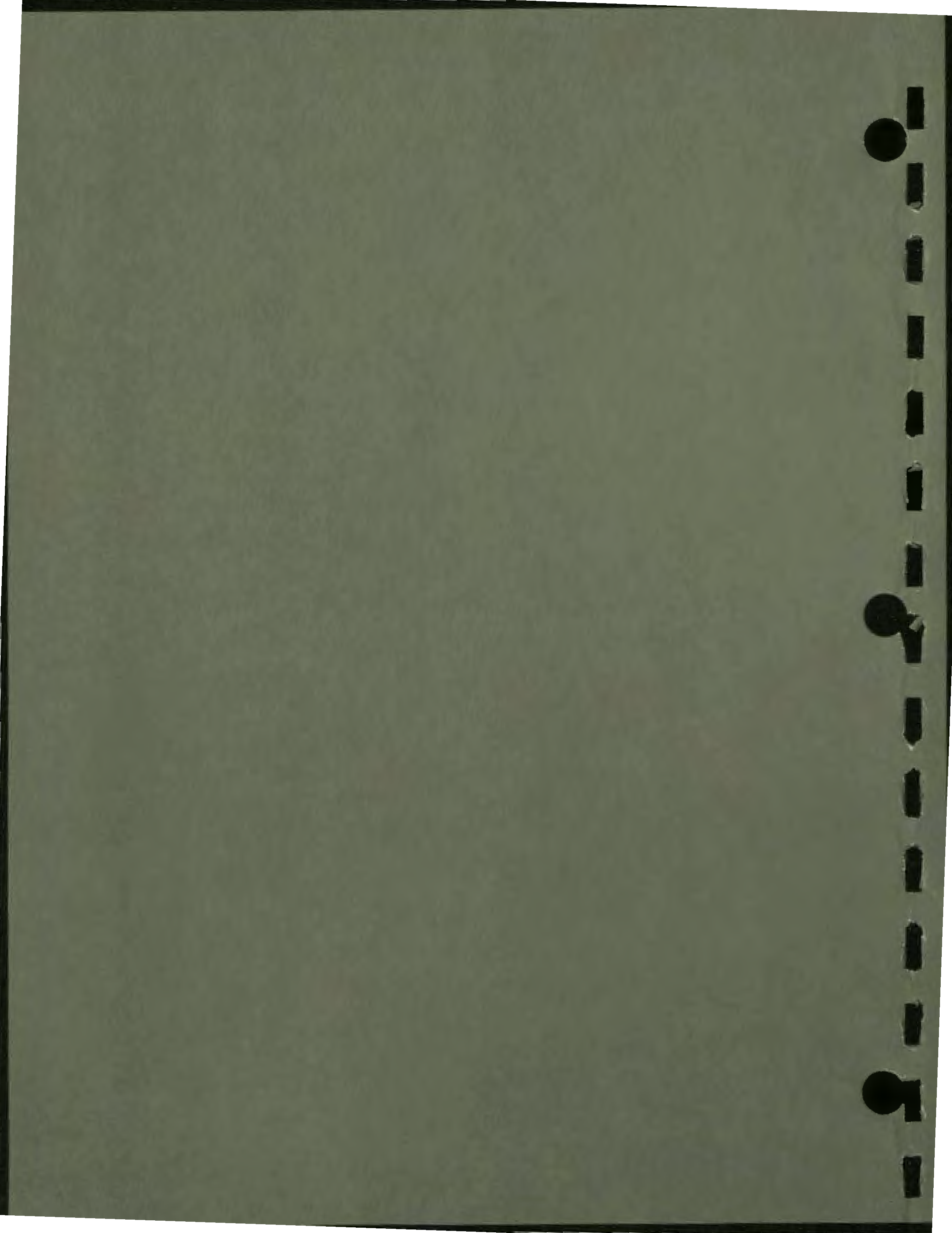
```
;
;         Display text and graphics with above characteristics.
;         Unit screen draws a shaded curve, an unshaded circle, and
;         two boxes in which the characteristics are written.
;
DO        screen
WRITE     state 1: bcolor blue–fcolor white
          state 2: sref 500,400–pattern dash
          state 3: italics 20–rorigin 600,100
AT        2300
WRITE     Press RETURN to change all the terminal characteristics.
PAUSE
;
;         Now change all characteristics.
;
BCOLOR    dark
FCOLOR    green
SREF
PATTERN   SOLID
ITALICS   0
RORIGIN   300,0
;
;         Display text and graphics with new characteristics.
;
DO        screen
WRITE     bcolor dark–fcolor green
          shading disabled–pattern solid
          italics 0–rorigin 300,0
AT        2300
WRITE     Press RETURN to restore terminal state number 1.
PAUSE
;
;         Restore first state, and show that the characteristics in
;         that state have been restored.
;
RESTORE   1
DO        screen
WRITE     state 1: bcolor blue–fcolor white
          shading disabled–pattern solid
          italics 0–rorigin 300,0
AT        2300
WRITE     Press RETURN to restore terminal state number 2.
PAUSE
;
;         Restore the second state, and show that the characteristics
;         in that state have been restored.
;
```

```
RESTORE   2
DO        screen
WRITE     state 1: bcolor blue–fcolor white
          state 2: sref 500,400–pattern dash
          italics 0–rorigin 300,0
AT        2300
WRITE     Press RETURN to restore terminal state number 3.
PAUSE
;
;         Restore the third state, and show that the characteristics
;         in that state have been restored.
;
RESTORE   3
DO        screen
WRITE     state 1: bcolor blue–fcolor white
          state 2: sref 500,400–pattern dash
          state 3: italics 20–rorigin 600,100
PAUSE

UNIT      screen
;
;         Create a graphics design on which to write the
;         descriptions of the terminal states.
;
ERASE
;
;         First draw a curve.
;
CURVE     400,300;440,320;500,380;510,300;590,400;550,430;490,460;430,400;
          400,300;400,300
AT        100,300
WRITE     curve               400,300;440,320;500,380;
                              510,300;590,400;550,430;490,460;
                              430,400;400,300;400,300
SREF                          $$ Disable shading so that subsequent
;                             graphics do not get shaded.
;
;         Next draw a box.
;
BOX       100,100;200,200
AT        100,70
WRITE     box 100,100;200,200
;
;         Then draw a circle around a relative coordinate.
```

```
;
RCIRCLE   0,0:25
RAT       0,0
WRITE     rcircle 0,0:25
;
;         And finally, draw a box to hold a description of the
;         terminal state's save/restore status.
;
BOX       739;1279:5
AT        840                 $$ Coordinate at which to
;                             begin writing description.

ENDLESSON
```

8

Response Judging



8

Response Judging

The discussion of response judging in Chapter 4 explains the default matching process and the default sequence of events when a student response matches a specified answer. The simple lesson in Chapter 5 shows some examples of response judging. This chapter describes the VAX DAL instructions that modify response judging.

There are three main ways you can modify the default response judging sequence.

- You can modify what the student must type to match the specified answer.
- You can modify the right and wrong judgments and what happens after a response matches.
- You can modify the way you specify right and wrong answers by specifying variables instead of the actual words the student types. With this modification, you can also specify that the answer must include units of measurement and that a number need not be exact.

MODIFYING WHAT A RESPONSE MATCHES – THE SPECS INSTRUCTION

The SPECS instruction modifies the exactness with which the student's response must match the specified right or wrong answers. With the default specifications, students can use any combination of uppercase and lowercase letters and any punctuation. They cannot insert extra words or use the right words in a different order. They cannot use expressions or units. The spelling tolerance test is applied to each word in the response.

Depending on the subject matter of the lesson, you can select the appropriate set of arguments to modify these default conditions. There is one set of current specifications. When the SPECS instruction changes a specification, the new specification is in effect for all subsequent response judging.

Arguments to the SPECS Instruction

The SPECS instruction has 11 sets of possible keywords. Each set controls a response-judging feature that can be turned on or off. The sets of keywords are:

- ANYORDER/NOANYORDER

  When ANYORDER is specified, the words in the student's response are compared to the words in the arguments to the RIGHT instructions. If all the words are the same regardless of word order, the response is judged right. The default is NOANYORDER.

- CAPS/NOCAPS

  When CAPS is specified, the student's response must be capitalized exactly as the argument to RIGHT to be judged right. When NOCAPS is specified, capitalization is not considered in judging. The default is NOCAPS.

- CONV/NOCONV

  When CONV is specified, students can use any unit conversion formulas defined by the author. When NOCONV is specified, all previously defined conversion formulas become unavailable: the units of measure the students use must match those used in the author's specified answers. The default is CONV.

- EXACT/NOEXACT

  When EXACT is specified, the spelling of all words in the student's response must be exactly as specified. The spelling tolerance test is not used. The default is NOEXACT.

- EXP/NOEXP

  When EXP is specified, students can use student variables in their responses. The student variables are evaluated when the response is judged. When student variables are allowed, RIGHTV and WRONGV must be used instead of RIGHT and WRONG. The default is NOEXP.

- EXTRA/NOEXTRA

  When EXTRA is specified, the student's response can have extra words and still be judged right. The default is NOEXTRA.

- FUNCT/NOFUNCT

  When NOFUNCT is specified, students cannot use system functions in their responses. They cannot, for example, respond to a question that asks for the square root of a number by using functions to calculate the square root. The default is FUNCT, which permits students to use system functions in their responses.

- MACHTOL/NOMACHTOL

  When MACHTOL is specified, a tolerance of plus or minus one ten-millionth of the value of the author's specified answer is allowed in student responses. Use MACHTOL with RIGHTV/WRONGV instructions when the author's real number answer is defined as a calculated expression, and a tolerance is not explicitly stated. MACHTOL allows for a discrepancy between the author's answer and a student's response that is within the range of what can be attributed to machine error. MACHTOL applies only to RIGHTV/WRONGV instructions. NOMACHTOL turns off the use of the MACHTOL feature, and is the default.

- PRECISE/NOPRECISE

  When PRECISE is specified, the student's response must precisely match one of the string answers designated in a RIGHTV or WRONGV instruction. PRECISE applies to RIGHTV/WRONGV instructions only, and supersedes all other SPECS instructions. When NOPRECISE is specified, the PRECISE rule is turned off, and any other current SPECS take affect. The default is NOPRECISE.

- PUNC/IMPUNC/NOPUNC

  When PUNC is specified, the punctuation in the student's response is checked against the punctuation in the argument to the RIGHT instruction. Wherever the author has placed a punctuation mark in the answer, the student must also place a punctuation mark in the response. The student's mark does not need to match the author's mark unless the SPECS EXACT instruction is also in effect. When IMPUNC is specified, punctuation in the student's response must match the punctuation in the RIGHT answer exactly, whether or not the SPECS EXACT instruction is in effect. When NOPUNC is specified, punctuation in the answer and in the response is ignored. The default is NOPUNC.

- UNITS/NOUNITS

  When UNITS is specified, students include the name of units of measurement in their responses when units of measurement are specified in the answers. The RIGHTV and WRONGV instructions must be used, and the CONVERT instruction can be used to specify that values are to be converted to an equivalent value in another unit. NOUNITS is the default.

Using the SPECS Instruction – ANYORDER and EXTRA Keywords

The sample lesson Icecream draws an ice cream cone in a color that is suitable for the flavor the student requests. The lesson Icecream is in Appendix F.

The challenge for the author of this lesson is to provide as many right answers as possible so that most flavors students enter display plausible colors of ice cream. (The use of the SYN instruction for this purpose is discussed later.)

The specifications are changed at lesson level to provide the most flexibility. Lemon and peppermint are right answers. With the EXTRA specification, responses such as lemon sherbet or peppermint twist are also judged right. As long as one of the words in the response is in the list of synonyms, the response matches.

With the ANYORDER specification, responses such as chocolate mint or mint chocolate chip are both right. These two answers, however, produce different colors. Mint and chocolate are both right answers. Response judging ends when the response matches a specified answer, so mint chocolate chip ice cream is green with dark chips. Chocolate mint ice cream is dark.

The other specifications have not been changed. There is no reason for requiring capital letters or punctuation in the responses, and using the default value of NOEXACT allows for typing errors.

MODIFYING WHAT A RESPONSE MATCHES – THE SYN INSTRUCTION

The SYN instruction specifies a target word that is used in the argument to RIGHT or WRONG, and a list of synonyms for the target word. You can add words to the lists of synonyms, or delete words at any point in the lesson. The lists of synonyms are available to all units.

The syntax of the SYN instruction is:

```
SYN      + "target","syn1","syn2","syn3"{,...}
```

The plus sign (+) indicates that this is a list of added synonyms. The first word following the plus sign is the target word, and the words that follow are the synonyms. Later in the lesson you can use the SYN instruction with a plus sign and the same target word to add to the list of synonyms. The added words are then synonyms for subsequent units.

You can also delete some words from the list or delete the entire list. To delete some words, use a minus sign (–) before the target word; then list the synonym or synonyms to be deleted. Other words in the synonym list are not affected. To delete the entire list, precede the target word with a minus sign, but do not list any words after the target word.

To indicate that a synonym list should be checked during response judging, you identify the target word in the argument to RIGHT or WRONG by enclosing it in double angle brackets, like this:

```
RIGHT     <<target>>
```

The lesson Icecream shows one use of synonyms in response judging. Each of the eight colors is specified as a target word at the beginning of the lesson. The synonyms for each color are flavors that look like that color. Then in the arguments to the RIGHT instructions, each color is specified as a right answer and enclosed in double angle brackets. When the student responds with a flavor such as lemon, the lesson searches the lists of synonyms and equates lemon with yellow.

The following example shows the SYN instruction and the RIGHT instruction for yellow.

```
SYN       +"yellow","lemon","banana"

QUERY
RIGHT     <<yellow>>
          FCOLOR  YELLOW
```

Synonym checking occurs only when the target word is enclosed in double angle brackets. If the word yellow appears in an answer without the angle brackets, the synonym lists are ignored, and only the response yellow is judged right.

MODIFYING WHAT THE RESPONSE MATCHES – THE NOISE INSTRUCTION

The NOISE instruction specifies that certain words are to be ignored in certain positions in a response. Noise words are specified with the NOISE instruction; then double angle brackets in the answer following RIGHT or WRONG specify the position where noise words are acceptable.

The syntax of the NOISE instruction is:

```
NOISE     + "noise_word","noise_word"{,...}
```

The plus sign (+) indicates that these words are to be added to the list of noise words. A minus sign (–) can be used to delete words, or to delete the entire list if no words follow the minus sign.

There is only one list of noise words. The list can be modified during execution of the lesson so that different words are acceptable as noise words in different student responses.

The following example shows noise words defined by the NOISE instruction and locations for noise words specified in a RIGHT instruction.

```
NOISE    + "the", "a", "that", "an", "this"

QUERY
RIGHT    <<>> man and <<>> dog
```

Assuming that the noise words listed above are in effect, the following responses are judged right:

```
man and dog
a man and a dog
the man and the dog
a man and this dog
that man and a dog
```

The following responses are judged wrong:

```
a man or a dog
a man and his dog
a tall man and a little dog
```

MODIFYING JUDGMENT OF THE RESPONSE – THE JUDGE INSTRUCTION

The arguments to the JUDGE instruction change what happens after a right or wrong response is matched. When a RIGHT or RIGHTV is matched, the default sequence is to process any response-contingent instructions, then go to the ENDQ and proceed from there. When a WRONG or WRONGV is matched, the default sequence is to process any response-contingent instructions, erase the response echoed when the student typed it, display the prompt again, and collect another response.

The JUDGE instruction and its argument are part of the response-contingent instructions, and apply only when that response is matched.

Arguments to the JUDGE Instruction

There are six possible keywords to JUDGE.

- AGAIN

  The AGAIN keyword causes judging to begin again at the start of the response-judging block. The same response is judged. The AGAIN keyword can be used in the response-contingent instructions with either RIGHT or WRONG.

- CONTINUE

  The CONTINUE keyword causes judging to continue. Judging does not stop when this answer matches the response. Instead, the response is compared to the next specified answer. The instruction that specifies the last matching answer controls the next step. If the last matching answer is the argument to a RIGHT instruction, the lesson continues after the ENDQ instruction. If the last matching answer is the argument to a WRONG instruction, the response is erased; and the lesson waits for a new response.

  The response-contingent instructions for all matching answers are executed.

- IGNORE

  The IGNORE keyword erases the student's response and displays the prompt again. Judgment then begins again with the new response. This keyword can be used after either RIGHT or WRONG instructions.

- NO

  The NO keyword reverses a right judgment. The student's response is erased, the prompt is displayed again, and the lesson waits for a new response. This keyword also reverses all the system variables associated with right and wrong responses.

- OK

  The OK keyword reverses a wrong judgment. The lesson goes to the ENDQ and proceeds from that point. This keyword also reverses all the system variables associated with right and wrong responses.

- STOP

  The STOP keyword stops judging and causes the lesson to go to the ENDQ immediately. Any response-contingent instructions following the STOP are not executed.

Using the JUDGE Instruction – the CONTINUE Keyword

The lesson Icecream in Appendix F shows a typical use of the CONTINUE keyword with the JUDGE instruction. Since a number of flavors include the word chip, chip is specified as a RIGHT answer. When this word occurs anywhere in a student's response, a flag is set to show that this response occurred, and the instruction JUDGE CONTINUE is given. This forces the lesson to go on looking for another right answer to select a color.

The order of RIGHT and WRONG instructions in the response-judging block is important with the JUDGE CONTINUE instruction. The first specified answer is always compared to the response. Whether the other specified answers are compared depends both on the response and on JUDGE instructions.

In the lesson Icecream, the response always needs to be tested for chip, then tested for a color. After the response matches a color, no other answers should be checked. Therefore, the first RIGHT instruction specifies chip. The response-contingent instructions for this answer include JUDGE CONTINUE. The order of the other RIGHT instructions is not important.

Using the JUDGE Instruction – the STOP Keyword

The unit prac1 in the lesson Multiply (which can also be found in Appendix F) shows a typical use of the STOP keyword. The default sequence forces the student to keep responding until the response matches an answer you have specified as right.

The default sequence is overridden with a JUDGE STOP instruction following the WRONG instruction. Notice that the feedback display of the right answer when the student makes a mistake comes before the JUDGE STOP instruction. When the JUDGE STOP instruction is executed, the lesson goes to the ENDQ immediately. Any response-contingent instructions following JUDGE STOP are not executed.

THE RIGHTV AND WRONGV INSTRUCTIONS

The instructions RIGHTV and WRONGV, like RIGHT and WRONG, specify right and wrong answers, but do so in a different way. With RIGHT and WRONG, answers are specified literally as the words students type in their responses. With RIGHTV and WRONGV, answers can be specified as variables or expressions.

The lesson Multiply shows one use of the instruction RIGHTV to specify a right answer as the current value of a variable. Because the variable is an integer, the characters the student types are converted to an integer before the response and the answer are compared. If the student enters an expression, the expression is evaluated before the response and the answer are compared. (If the right answer to one of the multiplication problems is 10, 5+5 is judged right.)

The RIGHTV and WRONGV instructions can specify a tolerance as either a percentage or an absolute value. If the response is within the tolerance range, it is judged as matching. The RIGHTV and WRONGV instructions can specify that a unit of measurement is a required part of the answer. With units, you can also use the CONVERT instruction so that students can enter more than one unit.

The syntax of RIGHTV and WRONGV is:

```
RIGHTV    value{,tolerance,unit_name}
WRONGV  value{,tolerance,unit_name}
```

The instructions RIGHTV and WRONGV must have arguments. The specified value can be an expression, a variable, or a constant of any data type.

You can specify the tolerance as a percentage or an absolute value. To specify a percentage, use a number followed by a percent sign. To specify an absolute value, use a number. The unit name is a string constant or variable that defines the unit for the specified value. The response must contain the specified unit name to be judged right.

Several of the SPECS instructions apply specifically to RIGHTV and WRONGV instructions. SPECS PRECISE, for example, is used when a precise match is required between the student's string or calculated response and the author's specified answer. Unlike SPECS EXACT, SPECS PRECISE requires that capitalization and word spacing, in addition to the other criteria checked by SPECS EXACT, precisely match in the student response and the author's answer. This response-judging feature is useful when evaluating student string responses that are chemical or mathematical formulas.

Student Variables

Student variables are variables whose names students can enter in a response. The current value of the variable is used to evaluate the expression.

Letting students respond with an expression to be evaluated is useful in a number of situations. You may be teaching a subject that uses constants such as Planck's constant. Students can use the standard notation for such constants in their responses; the lesson can perform the arithmetic. You might want to write a lesson that imitates a calculator, and let students use this as a tool.

In the following example, students can use numbers, operators, and the word pi in their responses.

```
LESSON    circum
SIZE      2
SPECS     EXP                   $$ Let student use student variables.
DEFINE    pi:REAL,STUDENT       $$ Define pi as a student variable.
DEFINE    r:REAL                $$ Define variable for radius.
ASSIGN    r:=17.5
PROMPT    "= "                  $$ Change prompt char so student sees an
                                $$ equation.
ASSIGN    pi:=3.14159
AT        310
WRITE     If the radius of a circle is <<T,r,4,1>> cm,
          what is the circumference?
          Use the * symbol for multiplication.
          Use pi

QUERY     *                     $$ put "= " after c for equation
RIGHTV    2*PI*R,1%             $$ allow 1% tolerance
.         WRITE                 Right. Multiplying by pi gives <<s,responsev>>
WRONG
.         MARKUP
.         WRITE     Try again.
ENDQ
ENDLESSON
```

Students can type several responses: the answers 2*pi*17.5, pi*35, and pi*17.5*2 are all judged right. If they wish, students can even complete the multiplication by pi. They cannot, however, use the variable r in their responses because r is not defined as a student variable. They must enter a number for the radius.

Student variables must be defined at lesson level and can be used in any unit.

Figure 8–1 shows one response.

```
If the radius of a circle is 17.5,
what is the circumference?
Use the * symbol for multiplication.
Use "pi"

C = 2 * 17.5 * pi

    Right.
```

MR-S-2134-82

Figure 8–1
The RIGHTV Instruction

Specifying a Tolerance

The DAL code example above specifies a 1% tolerance for the student's response. Tolerances are used for two reasons. The first reason is to provide more flexibility for the student. If a student answering the query in the above example actually multiplies by the value of pi, the response matches only if the student uses the exact value of 3.14159 that you assigned to the variable pi. If a student multiplies by 3.14, the response is judged wrong. Therefore, a tolerance is needed.

The second reason involves the representation of real numbers in the computer. A full treatment of this topic is beyond the scope of this manual. Briefly, if the author specifies the real number answer as a expression and does not also specify a tolerance, all student responses, even correct ones, are apt to be judged wrong because of machine error. You must either use a tolerance with real numbers, or use the SPECS MACHTOL instruction, which permits the minute discrepancies that can be attributed to machine error. For more detailed information on this topic, find a reference that explains floating-point numbers.

You can specify the value and a tolerance without using variables, as shown in the following examples.

```
RIGHTV    2.17 *(10.0^5), 10
WRONGV(8.8*(10.0^-6)) + (12.5*(10.0^-4)),10^-6
```

In these two examples, the tolerance is an absolute number. When the student enters a response, the value of the expression in the answer is calculated. The student's response matches if it is within the range defined by the result plus or minus the tolerance.

Specifying Units

The third argument to RIGHTV and WRONGV specifies a unit name. When this argument is used and SPECS UNITS is in effect, students must include the unit name to match.

You can also allow students to enter one of several unit names and a value in the units they select. For example, there are many subjects where values are normally expressed in one of several units, depending on the size of the value. Distance in the metric system can be expressed in meters, kilometers, centimeters, or millimeters.

Suppose that the right answer to a problem is 2.5 meters. This can also be expressed as 250 centimeters. The CONVERT instruction specifies the formula for conversion between units. As you write the lesson, you perform calculations and specify answers in the basic unit. When students take the lesson, they can enter any combination of value and units.

If you want students to be able to use unit conversion formulas instruction in only some of their responses, you can turn off access to all previously defined conversion formulas with the SPECS NOCONV instruction. Disabled conversion formulas can be turned back on with SPECS CONV.

The syntax of the CONVERT instruction is:

```
CONVERT     base_unit = secondary_unit_expression
```

The base unit is the unit name specified as the third argument to RIGHTV or WRONGV. The value specified in the RIGHTV or WRONGV instruction must also be in these units. The secondary unit expression defines the arithmetic operation required to convert a value in the secondary units to a value in the base unit.

The instruction CONVERT m = cm /100 is interpreted as follows. If m is the unit name in the answer, the students can enter either m or cm in the response. If the student enters m, compare the value in the response to the value in the answer. If the student enters cm, divide the value in the response by 100; then compare the resulting value to the value in the answer.

The following short lesson illustrates the CONVERT instruction.

```
LESSON      convert
SPECS       UNITS
CONVERT     ft = in /12
CONVERT     ft = yd *3
SIZE        2
DO          quest1
DO          quest2
; End of lesson level.

UNIT        quest1
AT          101
WRITE       Relatively short distances can be
            measured in inches, feet, or yards.
            Use the unit names in, ft, and yd.
            A stick is 2 ft long. How long is
            it in inches?
QUERY
RIGHTV      2,1%,"ft"
.           MARKUP
.           PAUSE
WRONG
.           MARKUP
.           PAUSE
ENDQ

UNIT        quest2
WRITC       How long is the stick in yards?
QUERY
RIGHTV      2,1%,"ft"
.           MARKUP
.           PAUSE
WRONG
.           MARKUP
.           PAUSE
ENDQ

ENDLESSON
```

Figure 8–2 shows a student's responses to this lesson. Notice that the student entered 2/3 yd in the second unit. The expression 2/3 is converted to a real number before it is compared to the answer. The real number 0.666667 is stored in the variable RESPONSEV. The student can also enter a real number.

```
Relatively short distances can be
measured in inches, feet, or yards.
Use the unit names in, ft, and yd.

A stick is 2 ft long.  How long is
it in inches?
>24 in  OK


 How long is the stick in yards?
 >2/3 yd  OK
```

MR-S-2135-82

Figure 8–2
The CONVERT Instruction

MISCELLANEOUS DISPLAY INSTRUCTIONS

Two different VAX DAL instructions modify the display of queries. The PROMPT instruction changes the system prompt character, and the INPUT instruction accepts a response without invoking the response-judging features of QUERY.

The PROMPT Instruction

The PROMPT instruction changes the character displayed as the system prompt at each QUERY instruction. In the lesson Multiply which can be found in Appendix F, two prompts are used: the default system prompt > and the equal sign. The system prompt is used for questions about what the student wants to do. The equal sign is used when the student's response is the answer to a multiplication problem. The instruction that selects the equal sign is:

```
PROMPT "= "
```

The argument is a string constant, and can be any character or series of characters, including characters from an alternate character set. Notice that there is a space following the equal sign. This provides a space before the student's response.

If you wish to use a string from an alternate character set as the prompt, use the following instruction format:

```
PROMPT "string", char_set_name
```

You are required to load the alternate character set prior to using this instruction.

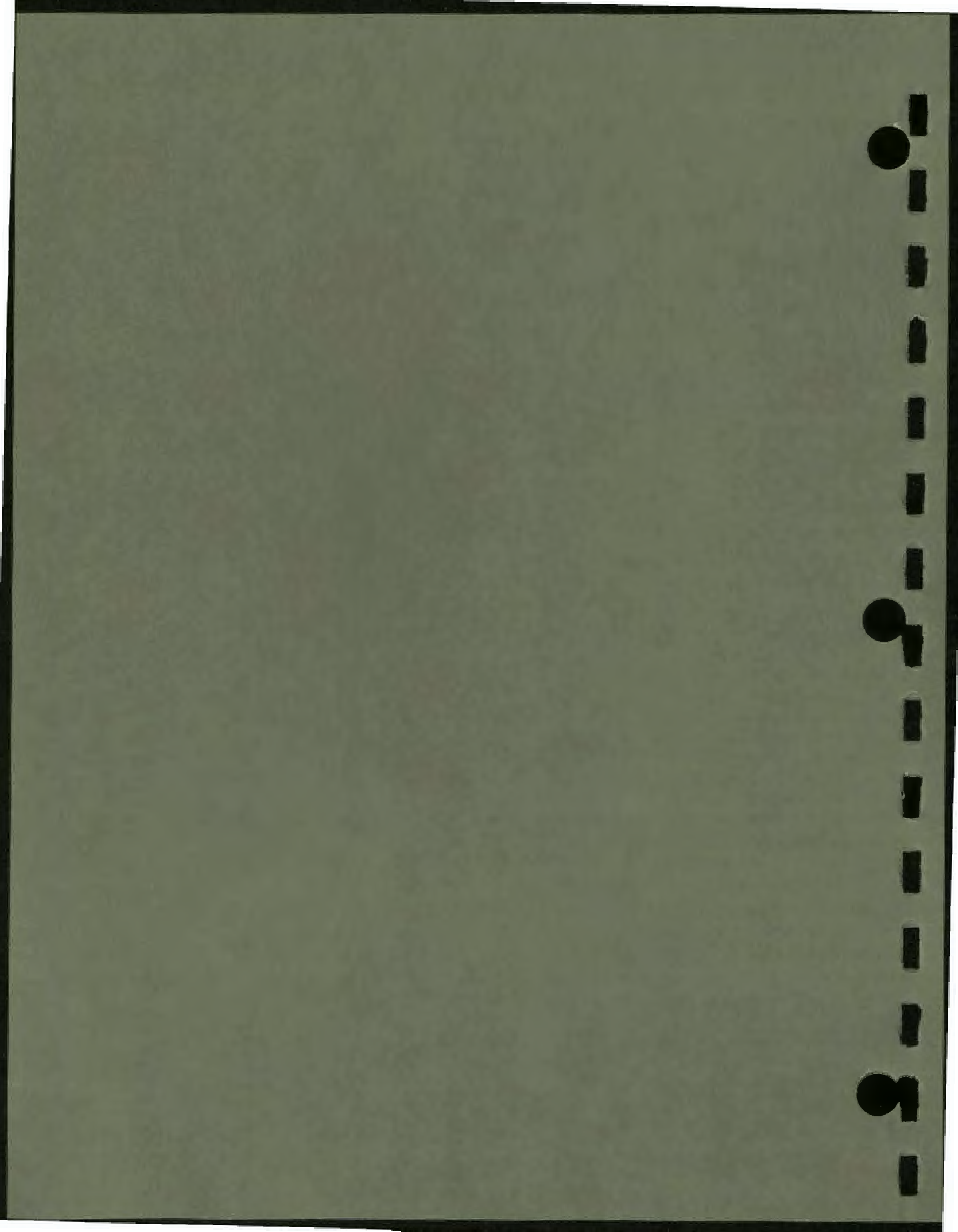
Once you have used the PROMPT instruction, the system displays the specified prompt until another PROMPT instruction is executed.

The INPUT Instruction

The INPUT instruction displays the system prompt character and waits for a student response. The response is stored in the system variable RESPONSE. The INPUT instruction is useful when you want the student to respond, but do not want the automatic judging and scoring of the response that the QUERY instruction initiates. A typical use for this instruction is following the display of a menu that the student selects from.

9

Response Processing



9

Response Processing

Chapter 8 discussed the instructions that modify default response judging. This chapter discusses the response-judging sequence itself, examining the different system variables that are updated to record student responses, errors, and scores.

Topics discussed in this chapter include:

- Scoring system variables
- Response-related system variables
- System variables in nested queries
- The ERRORV system variable

SCORING SYSTEM VARIABLES

VAX DAL maintains a number of system variables that contain student scores and information about student responses. Table 9–1 lists the scoring system variables discussed in this guide, what they contain, and how they are updated.

Table 9–1: Scoring System Variables

NAME	DATA TYPE	DESCRIPTION	CAUSE OF UPDATE
GOAL	integer 1 to 100	Number of current goal.	GOAL instruction
NNO	integer	Number of responses judged incorrect since beginning of lesson. Unless default response judging is overridden, this variable is 0.	Incremented by an incorrect response. Updated when a unit containing a QUERY block finishes.
NOK	integer	Number of responses judged correct since the beginning of the lesson.	Incremented by a correct response. Updated when a unit containing a QUERY block finishes.
NOKFIRST	integer	Number of correct responses made on the first attempt since the beginning of the lesson.	Incremented by a correct response on the first attempt. Updated when unit containing the QUERY finishes.
NUMTRIES	integer	Number of times the student has responded to the current QUERY.	Incremented after each response. Cleared when the unit containing QUERY begins.
QUERIES	integer	Number of QUERY instructions since the beginning of the lesson. QUERIES is not incremented when prompt is redisplayed after a wrong response. It is incremented if any instruction (REDO, LOOP, FOR, BRANCH) returns control to a point preceding the QUERY.	Updated by each new QUERY instruction.

Table 9–1: Scoring System Variables (Cont.)

NAME	DATA TYPE	DESCRIPTION	CAUSE OF UPDATE
SATISFIED	integer	Assigned the value 1 if the response is judged correct and 2 if the response is judged incorrect. If the unit does not contain a QUERY block, value assigned is zero.	Updated after each RIGHT, WRONG, RIGHTV, and WRONGV instruction. Cleared when a unit containing a QUERY begins.
SCORE	real	Student's total score. Default increment is 1. Increment is modified by argument to WEIGHT instruction.	Updated when a unit containing a QUERY finishes execution.
SCORES	real	Array of SCORES indexed by goal number. Array contains score for each goal. The index is an integer between 1 and 100. If any other index is used, the value –999 is returned.	Updated when unit containing QUERY finishes its execution.

The following is a step-by-step explanation of how DAL uses scoring system variables to record student scores.

When execution of a unit begins:
DAL sets the NUMTRIES and SATISFIED system variables to zero. The other scoring system variables do not update during the execution of the unit. If the unit does not contain a QUERY block, both SATISFIED and NUMTRIES still contain zero when the unit finishes.

Because both NUMTRIES and SATISFIED are reset to zero whenever a new unit starts, the current value of these two variables cannot be passed to another unit. Authors who need to pass the current value of NUMTRIES or SATISFIED must assign the value to a user-defined variable and pass the user-defined variable.

After the student enters the first response:
The response is counted as a try and DAL sets NUMTRIES to 1.

If the response matches an answer specified by a RIGHT instruction, DAL sets the SATISFIED variable to 1 and passes control to the ENDQ instruction.

If the response matches an answer specified as WRONG, or if the response does not match any of the specified answers, DAL sets SATISFIED to 2 and repeats the QUERY block. DAL adds 1 to the NUMTRIES variable for each additional try the student requires to answer the query.

By default, the response-judging sequence repeats until the student's response matches a RIGHT answer. When the student eventually enters a response that is judged correct, DAL sets SATISFIED to 1 and passes control to ENDQ.

After the ENDQ instruction, any subsequent instructions in the unit execute, and execution of the unit ends.

When execution of the unit ends:
DAL updates the scoring variables NOK, NOKFIRST, SCORE, and SCORES(GOAL) based on the values of the NUMTRIES and SATISFIED system variables. Because students remain by default in the response-judging block until they respond with a correct answer, SATISFIED always contains the value 1 when the execution of a unit finishes (unless default response judging is overridden).

If SATISFIED contains the value 1 when the execution of a unit finishes, DAL updates the scoring system variables in the following manner:

- 1 is added to NOK
- 1 is added to NOKFIRST if NUMTRIES = 1
- SCORE is updated based on WEIGHT
- SCORES(GOAL) is updated based on WEIGHT

In a unit that does not contain a QUERY block, the SATISFIED variable still contains the value zero when the unit finishes. The scoring variables do not update if SATISFIED contains zero when the unit ends.

How to override the default response-judging sequence:
By default, the response-judging sequence is repeated until the student enters a correct response. The JUDGE STOP instruction overrides the default response-judging sequence.

A JUDGE STOP instruction in a response-judging block immediately passes control to the ENDQ instruction, whether the student has matched a RIGHT answer or not. If the last response the student made before JUDGE STOP failed to match a correct answer, SATISFIED contains the value 2 when the unit finishes.

If SATISFIED contains the value 2 when the execution of a unit ends, DAL increments the NNO system variable. NNO contains the number of incorrect responses the student has made since the beginning of the lesson. An incorrect response does not affect the SCORE or SCORES system variables.

How to override the default scoring system:
By default, NOK, NOKFIRST, NNO, SCORE, and SCORES(GOAL) are updated only after execution of the unit ends. The SCORE UPDATE instruction overrides the default scoring variable update.

When executed in a unit, the SCORE UPDATE instruction updates the scoring variables immediately, based on the current value of the SATISFIED system variable. The update can occur only after the response-judging block in a unit. If you execute a SCORE UPDATE instruction before the unit finishes executing, you can use the updated scoring variables in the unit.

If you use SCORE UPDATE in a unit, the scoring variables do not increment again when the unit ends. DAL updates the scoring variables more than once for the same response-judging block only if the lesson executes the unit containing the block more than once with a REDO instruction.

Another way to modify the default scoring variable update is to assign a value to SATISFIED before execution of the unit ends. In this way, the scoring variables are updated when the unit ends based on the SATISFIED value the author assigns.

RESPONSE-RELATED SYSTEM VARIABLES

VAX DAL maintains a number of system variables that contain information about student responses, but are not used to keep score. Some of these variables can be used to restrict the way a query can be answered. Table 9–2 lists the most commonly used response-related system variables, and describes what they contain and how they are updated. Appendix C, which lists all system variables, includes other response-related variables.

Table 9–2: Response-Related System Variables

NAME	DATA TYPE	DESCRIPTION	CAUSE OF UPDATE
ERRORV	integer	Contains a numeric code that identifies the reason why a student's response failed to match an author's specified answer. See the discussion of ERRORV later in this chapter for a table of possible ERRORV values and their meanings.	Updated with each invocation of a RIGHT, WRONG, RIGHTV, or WRONGV instruction.
QELAPSED	integer seconds	Time available to answer the QUERY. QELAPSED counts down as time elapses. When time is up, the TIMEOUT variable is set.	Set to 0 (no limit) at the beginning of the lesson. Changed only if new value is assigned.
QLENGTH	integer	Number of characters allowed in response. If this number is reached, the response is terminated and judging begins.	Set to 0 (no limit) at beginning of lesson. Reset only if new value assigned.
RESPONSE	string maximum 500 chars	Student's exact response	New value assigned at QUERY or INPUT. New response is terminated by DELIMIT character or value of QLENGTH. Characters entered later are read by next INPUT or QUERY if typeahead is allowed

Table 9–2: Response-Related System Variables (Cont.)

NAME	DATA TYPE	DESCRIPTION	CAUSE OF UPDATE
RESPONSEV	real integer Boolean	String in RESPONSE evaluated as expression of same data type as arguments to RIGHTV or WRONGV.	Evaluated when RIGHTV or WRONGV is executed.
TIMEOUT	Boolean	Used with QELAPSED to determine if time allowed for response has occurred 0 = no timeout 1 = timeout	When QELAPSED is not 0, TIMEOUT is set to 0 at time of QUERY, to 1 when QELAPSED counts down to 0

DAL updates the RESPONSE variable each time a student enters a character string in response to a QUERY, INPUT, or PAUSE STRING instruction. RESPONSE contains the student's exact input.

If the response-judging block contains a RIGHTV or a WRONGV instruction, the contents of the RESPONSE variable are considered to be an expression. DAL evaluates the expression when the first RIGHTV or WRONGV instruction in the response-judging block begins execution. The value returned by the expression is stored in RESPONSEV.

Both RESPONSE and RESPONSEV are available in the unit containing the response-judging block.

By default, student responses can contain any number of characters and are terminated by the delimit character (RET).

The system variable QELAPSED defines a time limit for responding. You use the system variable TIMEOUT to test whether the time limit has elapsed. This can be useful for tests, for various kinds of simulation, and for games. The default value of QELAPSED is 0; this makes the QUERY available for an unlimited amount of time.

The following example gives the student five seconds to respond.

```
ASSIGN   QELAPSED:=5             $$ Wait 5 seconds for response.
QUERY
RIGHT
.                                $$ Response-contingent instructions.
WRONG
.        IF TIMEOUT =1
.        .        DO OUTIME
.        ENDIF
ENDQ
```

The ASSIGN instruction preceding the QUERY sets the value of QELAPSED to 5. When QELAPSED is set, TIMEOUT is also set to 0. Then, if the five seconds elapse before the student has typed the delimit character, TIMEOUT is set to 1; and whatever the student has typed is judged. If you test for the TIMEOUT value as part of the wrong answer processing, you can display an appropriate message to the student, handle the QUERY as an unanswered question, or to do whatever else is useful in this lesson.

The system variable QLENGTH defines the number of characters allowed in one response. After the student enters that number of characters, response judging begins. Like QELAPSED, QLENGTH must be assigned a new value before each QUERY to which it applies. The variable QLENGTH is reset to the default value of 0 at the ENDQ.

Menus provide one logical use for QLENGTH; another is in games where students always respond with the same number of characters or digits.

QLENGTH is useful, but should be employed with caution. Generally, it is easier for a student to do the same thing consistently. In lessons that use QLENGTH for some responses but not others, students can become confused about when they need to use the RETURN key and when they do not.

Suppose that the student gets used to pressing the RETURN key after every answer, and you then change QLENGTH and specify that responses are one character long. Whenever a key is pressed, the input is stored, and the stored characters are read the next time a QUERY, INPUT, or PAUSE instruction is encountered. If the student continues to press the RETURN key after an answer, the RETURN character is stored. Then, when the lesson comes to a QUERY, INPUT, or PAUSE instruction and reads what has been typed, this extra RETURN character is picked up. Depending on where the student is in the lesson, the (RET) character can be interpreted as a wrong answer, or can cause a display to be erased before it can be read. The default response sequence waits after the prompt character until the student types the system delimit character (RET).

USING SYSTEM VARIABLES

The following unit uses default response judging except for the three SPECS instructions. Modifications to this unit show some possibilities for using the scoring and response-related system variables in a lesson.

```
UNIT     capitol
AT       310
WRITE    What is the capitol of the United States?
SPECS    EXACT    $$ Spelling, punctuation, and cap letters count.
SPECS    CAPS
SPECS    PUNC
QUERY
RIGHT    Washington, D.C.
.        WRITE    Very good!
                  Press RETURN
.        PAUSE
WRONG    Washington
.        WRITE    Tell me the whole name.
                  Press RETURN and try again.
.        PAUSE
.        ERASE    510;680             $$ Erase the text on lines 5 and 6
;                                     before displaying the prompt again.
WRONG                                 $$ Any response not specified above.
.        WRITE    Be sure you are spelling and punctuating
                  the name correctly.
                  Press RETURN and try again.
.        PAUSE
.        ERASE    510;880
ENDQ
ERASE
```

One problem with the unit above is that it does not distinguish between a misspelled (or mistyped) response and a totally wrong response. If the student responds "washignton, DC", the response is judged wrong. The suggestion to check the spelling and punctuation is helpful. Exactly the same thing happens, however, if the student responds "Lima, Peru", and the suggestion is useless.

The following instructions are a revision of the unit capitol using the system variables to detect a misspelled or mistyped response. For this revision, the default specifications are in effect. Uppercase characters and punctuation are not considered in judging. The spelling tolerance test is applied.

```
UNIT      capitol
AT        210
WRITE     What is the capitol of the United States?
QUERY
RIGHT     Washington, D.C.
.         IF RESPONSE <> "Washington, D.C."
.         .         AT        615
.         .         WRITE     Check your spelling and punctuation.
                              Press RETURN and try again.
.         .         PAUSE
.         .         ERASE     615;880
.         .         JUDGE     NO
.         ELSE
.         .         AT        615
.         .         WRITE     Very good.
.         .         PAUSE     ELAPSED,3
.         ENDIF
WRONG
```

Because the default specifications are in effect, a misspelled, mispunctuated, or miscapitalized version of Washington, D.C. is judged right. Exactly what the student typed is stored in RESPONSE. The IF instruction compares the string "Washington, D.C." to the student's response. If the two strings are equal, the feedback following the ELSE is displayed; the answer is judged right; and the lesson goes to the ENDQ.

If the response is not equal to the string, the feedback about spelling and punctuation appears on the screen. This feedback appears only if students make mistakes in capitalization or punctuation, or if they misspell the word Washington. The instruction JUDGE NO reverses the right judgment, so the lesson erases what the student typed and redisplays the prompt. The ERASE instruction erases the feedback.

The WRONG Washington instruction from the original unit still makes sense, so it stays in the revised unit. But the WRONG (blank) instruction needs to be changed to help a student who really has no idea what the answer is. The NUMTRIES variable can help here.

```
WRONG
.       TEST      NUMTRIES
.       VALUE     1
.       .         AT        615
.       .         WRITE     That's not right.
.       VALUE     2
.       .         AT        615
.       .         WRITE     Let me give you a hint.
                            It was named for the first president.
.       VALUE     3
.       .         AT        615
.       .         WRITE     The answer is Washington, D.C.
                            Put this on your list of capitals to review.
                            Press RETURN to continue.
.       .         PAUSE
.       .         JUDGE     STOP
.       ENDTEST
.       AT        815
.       WRITE     Press RETURN to try again.
.       PAUSE
.       ERASE     615;1080
ENDQ
```

The TEST structure tests the value of the variable NUMTRIES. If the student's first response is wrong, the feedback is just that the response is wrong. On the second try, the lesson displays a hint. If the third try is still judged wrong, the answer is displayed. The JUDGE STOP instruction ends the judging and goes to the ENDQ.

The instructions following the ENDTEST are executed after the first two wrong answers. These instructions can be included in the instructions following VALUE 2 and VALUE 3, but this means writing them twice. They are skipped after the third time, because JUDGE STOP causes the unit to go to the ENDQ immediately.

The lesson Icecream in Appendix F shows another use of the RESPONSE variable. If the student's response is not one of the flavors of ice cream that the lesson recognizes, the lesson uses the RESPONSE variable to say that the flavor requested is not available. Of course, anything the student types is repeated, and it is possible to have nonsensical feedback.

SYSTEM VARIABLES IN NESTED QUERIES

A nested query occurs whenever a DO instruction in a QUERY block executes a unit that contains a QUERY block.

The DAL units below show the instructions that create a nested query.

```
UNIT      unita
AT        505
WRITE     In what year did Christopher Columbus
          set sail for what he thought was India?
QUERY     810
RIGHTV    1492
.         AT        1015
.         WRITE     Excellent!
.         PAUSE     ELAPSED,3
.         ERASE
WRONG
.         DO        unitb
.         JUDGE     STOP
ENDQ
RETURN

UNIT      unitb
AT        1015
WRITE     Sorry, but the correct answer is 1492.
          Here's a bonus question, however, and if you
          get this second question correct it will make
          up for missing the first question.
AT        1515
WRITE     What were the names of Columbus's three ships?
NOISE     +"and", "the"
SPECS     ANYORDER
QUERY     1715
RIGHT     <<>>Nina<<>>Pinta<<>>Santa Maria
.         AT        1915
.         WRITE     Excellent!
WRONG
.         AT        1915
.         WRITE     Sorry, but you're wrong again.
                    Time to read that chapter.
.         JUDGE     STOP
ENDQ
IF        SATISFIED = 2              $$ So the NNO variable isn't
.         ASSIGN    SATISFIED:=0     $$ incremented if the student
ENDIF                                $$ misses the bonus question.
PAUSE
ERASE
RETURN
```

In the above DAL code, a DO instruction in the QUERY block in unita executes unitb. Because unitb itself executes a QUERY block, the QUERY block in unitb is nested inside the QUERY block in unita. Unitb executes conditionally, depending on the outcome of response judging in unita.

Before a unit called from within a QUERY block starts, DAL saves the values of many scoring variables and response-related variables. DAL does this in order to preserve the status of the calling QUERY block. When the called unit finishes and control returns to the calling QUERY block, DAL restores the saved values to the scoring variables and response-related variables. In this way, the values of the variables in a QUERY block are not disturbed if the QUERY block executes a response-contingent unit.

The items in the table below are saved when a unit is executed from within a QUERY block.

Table 9–3: Items Saved when Queries are Nested

SATISFIED variable	NUMTRIES variable
RESPONSE variable	ERRORV variable
Response length	Response ECHO flag
Typeahead flag	Number of words in the response
Latency	ANSCNT variable
QLENGTH variable	QELAPSED variable
OKWORD variable	NOWORD variable
Current Goal	Current text size
Current prompt	Current prompt font
Prompt location	Flag on whether scores have been updated

The original values of the items listed above are restored when a unit called from a QUERY block finishes executing. DAL saves and restores system variable values only when DAL units are called from within QUERY blocks. Variable values are not saved when a QUERY block calls a routine written in another programming language. Also, variable values are not saved if control transfers to a condition unit.

With the exception of the SATISFIED and NUMTRIES variables, the current value of the variables listed above can be passed to a unit called from a QUERY block. However, changes made to the values in the called unit cannot be passed back to the QUERY block when the called unit finishes executing.

SATISFIED and NUMTRIES are reset to zero when the called unit starts its execution. Therefore, the current value of these variables cannot be passed. SATISFIED and NUMTRIES values can be passed if the called unit is a condition unit instead of a regular DAL unit. None of the items listed above are saved and restored if a condition unit is called from within a QUERY block.

The previous values of several scoring system variables are not restored when a QUERY block calls another unit. These variables are incremented to reflect the results of response judging in the called unit. These variables are:

NOK	NOKFIRST
SCORE	SCORES
NNO	QUERIES

USING THE ERRORV SYSTEM VARIABLE

DAL sets the ERRORV system variable each time a student response fails to match the answers specified by a RIGHT, WRONG, RIGHTV, or WRONGV instruction in a response-judging block. DAL sets ERRORV with a value that specifies the reason why the match failed. The table below lists what each ERRORV value means.

Table 9–4: ERRORV Values

[illegible]	[illegible]
0	Student response matches the answer string adequately.
10	General failure. Error not identified by other ERRORV codes.
11	Wrong word length.
12	Spelling algorithm failure.
21	Extra words in response. (Occurs only if the student's response otherwise matches the specified answer.)
31	SPECS PRECISE on; match failed.
41	Capitalization in student response does not match that in the specified answer.
51	SPECS IMPUNC on; punctuation in the student response does not match that in the specified answer.

Table 9–4: ERRORV Values (Cont.)

VALUE	MEANING
71	Unit of measure required; no unit in response.
72	Unit of measure in response does not match unit in the specified answer and cannot be converted.
73	Unit in response, no unit in specified answer.
74	SPECS EXACT on; unit in response matches either unit in answer or unit in answer by conversion, but neither exactly.
75	SPECS NOCONV on, unit in response correct by conversion, but use of conversion formulas not allowed.
91	Value of response is wrong.
92	Mismatched parentheses in response.
93	Unidentified variable in response.
94	Response is missing an operand.
95	Unidentified function in response.
96	Invalid argument to function.
97	Too few arguments for function.
98	Too many arguments for function.

Setting ERRORV Values

DAL sets ERRORV to zero whenever a RIGHT, WRONG, RIGHTV, or WRONGV instruction begins executing. If the student's response fails to match any of the answers specified by the instruction, DAL sets ERRORV with a nonzero value. Each RIGHT, WRONG, RIGHTV, and WRONGV instruction in a QUERY block can assign a different value to ERRORV. This means that ERRORV can potentially contain different values at different places in the same QUERY block.

ERRORV contains a value other than zero only if the response-matching process fails. If a response is judged wrong, ERRORV contains a zero because the response-matching process succeeded in matching the student's response to an answer specified by a WRONG instruction. A WRONG instruction without an argument always produces a zero in ERRORV, because the match always succeeds.

Checking ERRORV Values in a Response-Judging Block

Authors must use CHECKERR blocks to test the value of the ERRORV variable in a response-judging block. CHECKERR blocks are blocks of instructions that begin with the CHECKERR instruction and contain control logic structures. Position a CHECKERR block after the response-contingent instructions to any RIGHT, WRONG, RIGHTV, or WRONGV instruction. A CHECKERR instruction must be at the same level of dot indentation as the RIGHT, WRONG, RIGHTV, or WRONGV instruction it follows.

TEST structures and IF structures work well in CHECKERR blocks. These control logic structures can be used to execute error-contingent instructions if the match fails in an anticipated way. The example below illustrates how ERRORV might be used.

```
UNIT      rootprob
AT        610
WRITE     Use a system function to answer the following question:
AT        810
WRITE     What is half the square root of 50 multiplied by 4?
QUERY     1010
RIGHTV    (SQRT(50)/2)*4
.         AT        1520
.         WRITE     EXCELLENT!
CHECKERR
.         TEST      ERRORV
.         VALUE     0         $$ So "OTHER" is not executed
;                             $$ for a correct response.
.         VALUE     92        $$ (Mismatched parentheses)
.         .         AT        1520
.         .         WRITE     You have an uneven number of
                              parentheses in your response.
                              Press RETURN to try again.
.         .         PAUSE
.         .         ERASE     1010;2360
.         .         REDO
.         VALUE     94        $$ (Response missing operand)
.         .         AT        1520
.         .         WRITE     You left an operand out of
                              your response.
                              Press RETURN to try again.
.         .         PAUSE
.         .         ERASE     1010;2360
.         .         REDO
```

```
.       VALUE    95        $$ (Unidentified function)
.       .        AT        1520
.       .        WRITE     You may have misspelled the
                           function name. Use SQRT(50).
                           Press RETURN to try again.
.       .        PAUSE
.       .        ERASE     1010;2360
.       .        REDO
.       VALUE    97        $$ (Too few arguments to the function)
.       .        AT        1520
.       .        WRITE     You forgot to include an
                           argument in the function.
                           Press RETURN to try again.
.       .        PAUSE
.       .        ERASE     1010;2360
.       .        REDO
.       OTHER
.       .        AT        2055
.       .        WRITE     <<S,ERRORV>>         $$ report ERRORV values
;                                               $$ not responded to above.
.       ENDTEST
WRONG
.       AT       2020
.       WRITE    Sorry, but you're wrong.
                 Press return to try again.
.       PAUSE
.       ERASE    2020;2360
ENDQ
```

Figures 9–1 and 9–2 show displays produced by UNIT rootprob (listed above).

```
Use a system function to answer the following question:

What is half the square root of 50 multiplied by 4?

>(SQRT(50)/2*4



          You have an uneven number of
          parentheses in your response.
          Press RETURN to try again.
```

MR-S-3966-85

Figure 9–1
Using ERRORV to Detect Missing Parentheses

```
Use a system function to answer the following question:

What is half the square root of 50 multiplied by 4?

>(SRT(50)/2)*4


          You may have misspelled the
          function name.  Use SQRT(50).
          Press RETURN to try again.
```

MR-S-3967-85

Figure 9–2
Using ERRORV to Detect a Misspelled System Function

Separate CHECKERR blocks can test the value of the ERRORV variable after the response-contingent instructions to each RIGHT, WRONG, RIGHTV, or WRONGV instruction in a QUERY block. The ERRORV variable cannot be tested outside of a CHECKERR block. CHECKERR blocks cannot be used outside QUERY blocks.

Interpreting ERRORV Values

DAL analyzes responses that are expressions differently from responses that are strings. What follows is a description of the two processes DAL uses to analyze student responses and set ERRORV.

In a response consisting of several different elements, any or all elements in the student's response may fail to match elements in the specified answer. Note, therefore, that some ERRORV codes detect an overall student response problem. Other ERRORV codes detect an error in just one element in the response. ERRORV code 10 is the general purpose, catch-all flag that DAL sets when a match attempt fails, but no specific reason for the failure can be identified.

Interpreting ERRORV Values for Expressions

Student responses that fail to match answers specified by RIGHTV or WRONGV instructions can produce ERRORV code 10, or ERRORV codes higher than 51.

DAL analyzes responses that are expressions in the three consecutive phases listed below. A fatal error in any one of the phases terminates the analysis.

1 The structure of the expression is examined for format, existing variables, function arguments, and so on.
2 The unit of measure designation is examined (if SPECS UNITS is in effect).
3 The value of the expression as a whole is examined.

ERRORV reports the first error it encounters in an expression. If the expression contains two errors, the second is not detected. For example, the following expression has both an invalid argument to the function and unbalanced parentheses. The ERRORV code this expression produces is shown at the right of the expression.

```
30*(SQRT(-100) cm          96 (invalid argument)
```

If SPECS NOUNITS (the default)is in effect, then ERRORV codes from 71 to 75 are not applicable. Note that, if SPECS NOUNITS is in effect, students can enter anything or nothing as units of measure; DAL simply ignores units in the response.

When SPECS UNITS is on, however, ERRORV codes from 71 to 75 can be produced. If DAL detects a problem with units, analysis stops and the appropriate ERRORV code is set. Note that, if the match fails because of units of measure, DAL never examines the value of the expression. An incorrect response that uses the wrong units of measure produces a unit error code in ERRORV.

The examples below illustrate the ERRORV values that result from specific student responses. If the specified answer to a query is, or evaluates to, 300 cm, then a response from Column A below produces the ERRORV value listed in Column B.

Column A	Column B
30*SQRT(100) cm	0 — response is correct
30SQRT(6) cm	72 — incorrect units (operator missing)
30*SQRT(6) cm	91 — value of response is incorrect
(300)) cm	92 — mismatched parentheses
((300) cm	92 — mismatched parentheses
30 * cm	93 — undefined variable
30** 15 cm	94 — operand missing
30*SINT(45) cm	95 — unidentified function
30*(SQRT(-100)) cm	96 — invalid argument to the function
30*(SQRT()) cm	97 — too few arguments to the function
30*SQRT(5,6) cm	98 — too many arguments to the function

Interpreting ERRORV Values for String Responses

The ERRORV codes for string responses are largely self-explanatory. For example, ERRORV code 51 (SPECS IMPUNC in effect; incorrect punctuation in response) is produced when SPECS IMPUNC is in effect and the response is incorrectly punctuated. However, things become more complicated when a response contains more than one error or more than one word. What follows is a description of the phases of analysis DAL applies to string responses.

Analysis of Single Word Responses

DAL first checks the length of the response. If the response contains too many or too few characters to match the specified answer, the response is assumed to be the wrong word. Analysis terminates and ERRORV is set to 11 (meaning wrong word length). If the response contains other errors, they are not detected.

If SPECS NOEXACT (the default) is in effect, the response and the specified answer can differ in length by one or more characters, depending on the length of the answer. If SPECS EXACT is in effect, the response and the answer must have exactly the same number of characters. When SPECS EXACT is in effect, incorrect punctuation often produces the word length code.

If word length is acceptable, DAL checks the capitalization in the response. If SPECS CAPS is in effect, incorrect capitalization terminates the analysis and sets ERRORV to 41 (meaning incorrect capitalization).

If the response is adequately capitalized, DAL compares the spelling of the response with the spelling of the specified answer. If SPECS NOEXACT is in effect, the response must pass the default spelling tolerance test, but the response does not need to exactly match the specified answer. SPECS EXACT requires that spelling in the response exactly match the spelling in the answer. If the response does not pass the spelling test, incorrect spelling terminates the analysis and sets ERRORV to 12 (spelling algorithm failure).

If the response passes all three phases of the analysis described above, the response matches one of the specified answers, and ERRORV remains set to 0 (indicating a successful match between the response and an answer).

If SPECS PRECISE is in effect, DAL sets ERRORV to 31 (SPECS PRECISE in effect; match failed) if the match fails for any reason.

If a RIGHT, WRONG, RIGHTV, or WRONGV instruction specifies more that one answer, ERRORV reflects the comparison of the student's response to the last answer in the string of specified answers.

Analysis of Multiword Responses

All of the conditions described above are true of responses that consist of one word. Student responses that consist of several words are handled somewhat differently. The following example illustrates this. Let the correct answer to a query be:

McRae of the USA

The following student responses produce the corresponding ERRORV codes if SPECS CAPS and SPECS EXACT are both in effect:

McRae of the Usa 41 (CAPS)

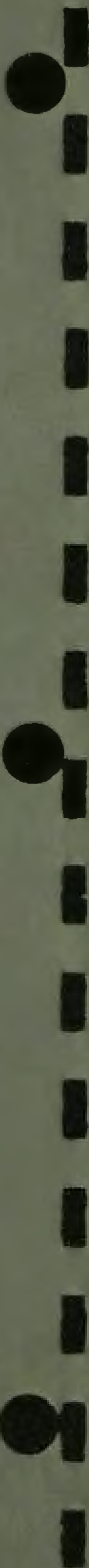
McRAE of the USA 11 (Wrong word length)

In the first instance, the incorrect capitalization of USA is the only thing wrong with the response. It also is a problem with the last element in the response. Hence, ERRORV code 41. In the second instance, "McRAE" fails to match "McRae" because of SPECS CAPS, so DAL looks for another word in the answer that matches "McRAE". DAL attempts to match the student's "McRAE" against the author's "of", "the", and "USA". The judgment of "McRAE" against "USA" is that word length is too long, hence, ERRORV code 11.

When a word in a student's response fails to match a word in the specified answer, the student's incorrect word is checked against each subsequent word in the author's answer. ERRORV code 11 almost always results. ERRORV codes other than 11 are produced only if the incorrect word is the last word in the response.

10

Graphics



10

Graphics

This chapter discusses the use of the graphics instructions in DAL. Graphics can be used in many different ways to make a lesson more interesting. They can draw attention to a particular part of the display, both when you display it originally and after the student has responded. Graphics can be directly related to the subject matter of the lesson, as in geometry.

The following instructions are explained in this chapter:

- Basic Graphics Instructions: BOX, CIRCLE, CURVE, DOT, LINE, and VECTOR

  These instructions draw elements that can be combined into pictures or used to enhance and emphasize text displays.

- PATTERN

  The PATTERN instruction draws patterns such as dotted or dashed lines.

- SREF

  The SREF instruction shades areas of the screen so that they appear in a solid or patterned color.

- Text Enhancement Instructions: ITALICS and TROTATE

  The ITALICS instruction displays characters with vertical lines on a slant. The TROTATE instruction displays text with the baseline at an angle across the screen.

- ERASE

  The ERASE instruction erases the entire screen or a specified rectangle on the screen.

- MODE

  The MODE instruction modifies the display mode of both text and graphics, and controls features such as blinking, reversal of background and foreground color, overlaying characters, and writing in erase mode.

- Relative Graphics

  Relative graphics change the 0,0 location from the top left corner to any point on the screen, then specify addresses that are relative to the new 0,0 location. The relative graphics instructions RAT, RBOX, RCIRCLE, RCURVE, RDOT, RLINE, and RVECTOR correspond to the basic graphics instructions. Relative graphics can also be rotated, and their size can be modified independently in the x and y directions.

This chapter also discusses the graphics system variables.

Chapter 3 explains screen addresses, text display, and the selection of background and foreground colors. Examples in this chapter use the graphics instructions and the AT, WRITE, SIZE, BCOLOR, and FCOLOR instructions.

BASIC GRAPHICS INSTRUCTIONS

The basic graphics instructions draw line figures. You can combine these figures to draw pictures, or use them to emphasize or isolate parts of the display. The instructions are named for the figures they draw:

- BOX draws a rectangle. The sides can be more than one line thick.
- CIRCLE draws a circle or an arc.
- CURVE draws a curve between points on the screen. Any number of points can be included.
- DOT draws a dot.
- LINE draws a straight line.
- VECTOR draws an arrow.

Figure 10–1 shows a picture drawn with the basic graphics instructions.

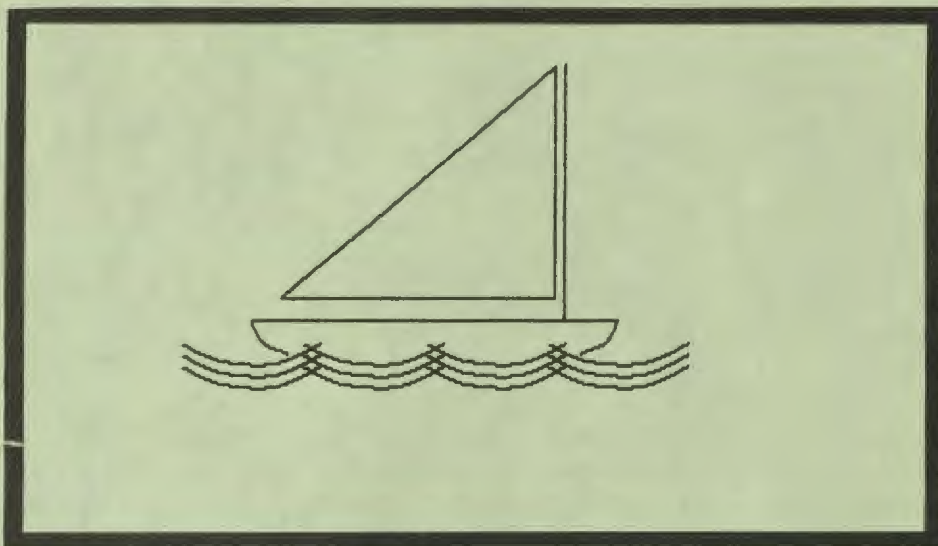


Figure 10–1
Basic Graphics Instructions: Sailboat

The following lesson draws Figure 10–1.

```
LESSON   boat
DEFINE   x,y : INTEGER
ERASE
BOX      0,0;0.999,0.999:–15  $$ frame
FCOLOR   GREEN
LINE     200,280;500,280      $$ deck
LINE     456,50;456,280       $$ mast

FCOLOR   WHITE
LINE     448,55;225,260       $$ sail
LINE     225,260;448,260
LINE     448,260;448,55

FCOLOR   BLUE
FOR      y := 220,240,10      $$ waves
.        FOR                  x:=200,500,100
.        .                    CIRCLE x,y:100,235,305
.        ENDFOR
ENDFOR
```

```
FCOLOR GREEN              $$ bottom of
AT     200,200            $$ boat
CURVE  200,280;230,310;230,310
AT     500,200
CURVE  500,280;470,310;470,310
PAUSE
ENDLESSON
```

BOX

The BOX instruction draws a rectangle. An optional argument designates the line thickness for the sides of the box. The syntax is:

```
BOX     {corner;}opp_corner{;thickness}
```

The first two arguments are the screen addresses of the two opposite corners. The first corner address, which specifies the upper left corner of the box, is optional. If it is not specified, the box begins at the current location.

The BOX instruction that draws the frame around the picture of the sailboat above uses the addresses 0,0 and 0.999,0.999. The fine coordinates for the first address select the upper left corner of the screen. The normalized coordinates for the second address select the lower right corner.

The thickness argument, which is optional, specifies the thickness of the sides of the box in dots. If thickness is not specified, the lines for the four sides of the box are one dot wide. If thickness is a positive number, the additional lines are drawn outward from the addresses. If thickness is a negative number, the lines are drawn inward. In the lesson that draws Figure 10–1, the thickness of the frame is –15, so the lines are drawn inward from the limits of the screen.

You can specify the screen addresses in any of the three forms: row and column, fine, or normalized. Thickness is always specified in screen dots.

Figure 10–2 shows some boxes and the instructions that draw them.

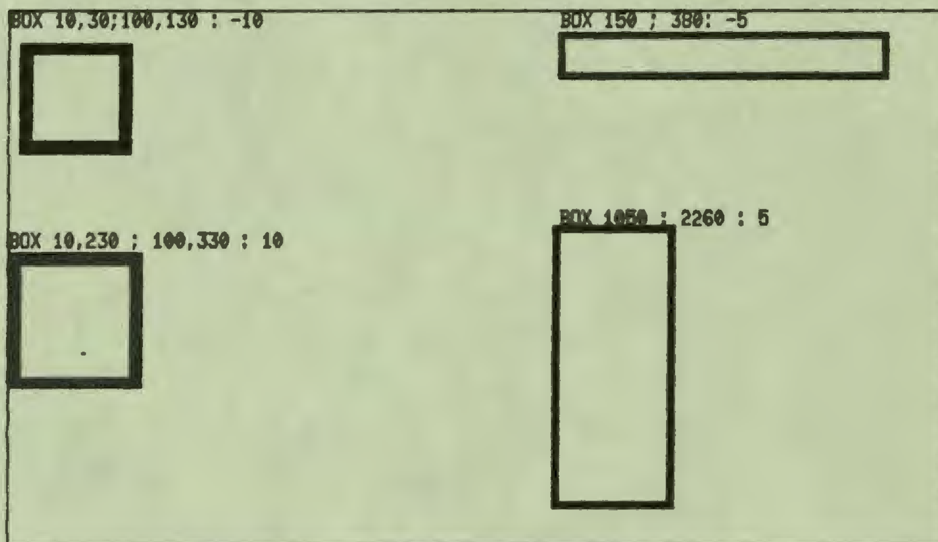


Figure 10–2
The BOX Instruction

CIRCLE

The CIRCLE instruction draws either a circle or an arc. The syntax is:

```
CIRCLE    center:radius{,start_of_arc,end_of_arc}
```

The first argument is the location of the center of the circle in any of the three forms of screen addresses. The second argument is the radius in dots. The following instructions draw three concentric circles.

```
CIRCLE    350,250 : 150
CIRCLE    350,250 : 100
CIRCLE    350,250 : 50
```

To draw an arc, use the third and fourth arguments to specify the starting and ending points of the arc. These arguments are in degrees. Zero degrees is at 3 o'clock. Other points can be specified as positive or negative. Twelve o'clock, for example, is either 90 or –270. The arc is drawn counterclockwise from the starting point to the ending point.

Figure 10–3 shows four circle instructions and the circle and arcs they draw.

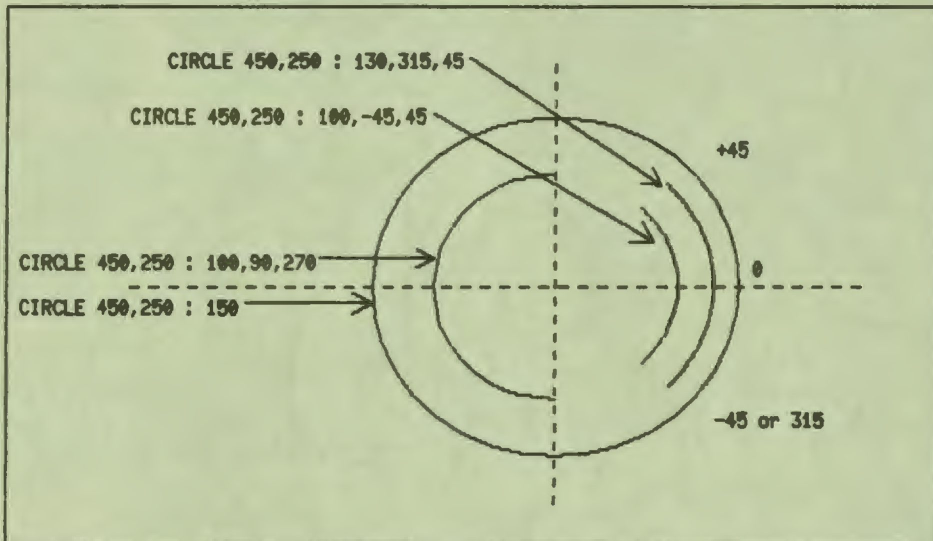


Figure 10–3
The CIRCLE Instruction

Look at the instruction that draws the 180-degree arc on the left. The starting point is 90 degrees, and the arc is drawn counterclockwise to 270 degrees. Reversing the order of the arguments draws the arc on the other side: that is, from 270 degrees counterclockwise to 90 degrees.

The waves in the sailboat picture are drawn with the CIRCLE instruction. The radius and arc arguments are the same for each instruction. The address of the center changes for each of the 12 arcs.

CURVE

The CURVE instruction draws a curve. It requires at least three addresses, and more can be specified. The syntax is:

```
CURVE    address;address;address{;...;address}
```

When three addresses are specified, the curve is drawn between the first and second points, as follows. Four points are used for curve fitting, which determines the shape of the curve. The current location (not specified as an argument) determines the beginning slope of the curve. The curve is drawn between the first two specified points. The fourth point (the last address specified) determines the ending slope of the curve.

Figure 10–4 shows examples of curve fitting. The solid lines represent the curves actually drawn with the CURVE instruction. The vertical dashed line begins at the current location that was set before the CURVE instruction. The horizontal dashed lines are drawn from the second-to-last to the last positions specified with the CURVE instruction. The curve that is drawn is a segment of the curve that passed through all four points.

The two curves in this figure begin and end at different absolute locations, but the distance between the first and second points specified is the same. The difference in the appearance of the curves is caused by the difference in the current location and the difference in the distance between the second-to-last point and the last point.

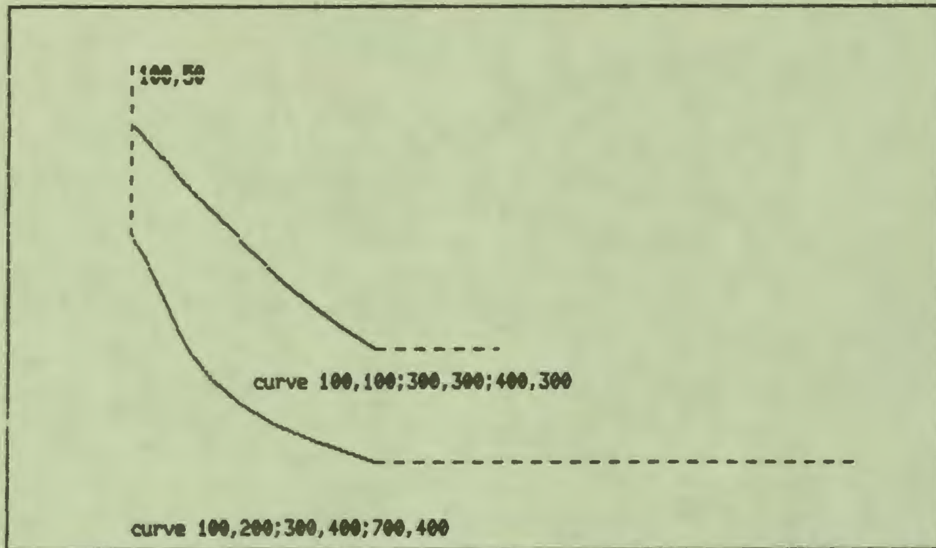


Figure 10–4
The CURVE Instruction: Three Points

Figure 10–5 shows two curves, each drawn by specifying four addresses. The difference in appearance of these two curves is again caused by the different current location before each CURVE instruction, and the different last location specified in each instruction.

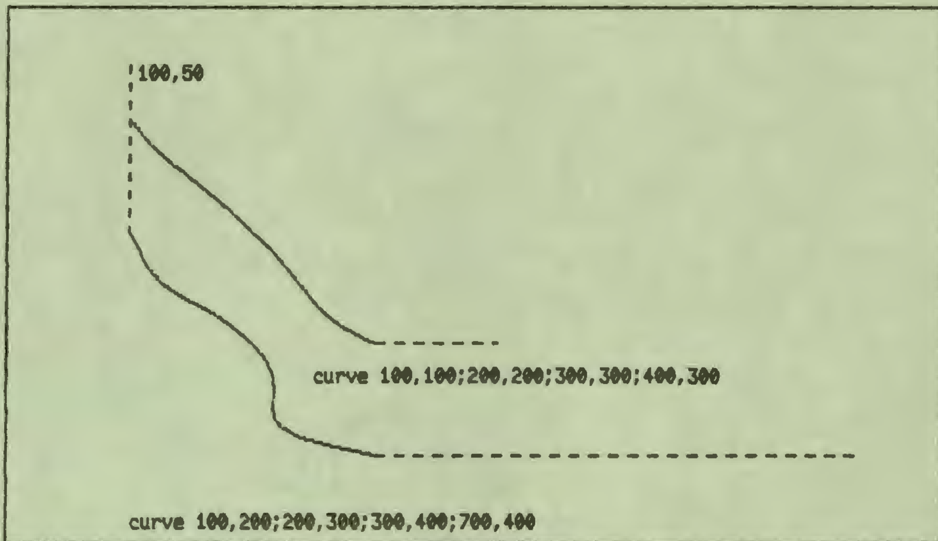


Figure 10–5
The CURVE Instruction: Four Points

When more than three addresses are specified, the curve is drawn in segments. The first segment is drawn between the first two addresses. The current location and the third address are used for curve fitting. The second segment is then drawn between the second and third addresses; the first and fourth addresses are used for curve fitting. This process continues for the curve between any number of addresses.

The CURVE instructions used in the sailboat picture are:

```
AT      200,200                  $$draw curves for
CURVE   200,280;230,310;230,310  $$boat
AT      500,200
CURVE   500,280;470,310;470,310
```

The horizontal line for the deck of the boat is drawn from 200,280 to 500,280. These two locations are also the first locations in the CURVE instructions. The AT instruction preceding each CURVE instruction sets the current location 80 dots above the first specified point. Because of the current location, the starting slope of the drawn curve is steep. The second location specified in both instructions is 30 dots down and 30 dots in toward the center of the screen. The curve ends at this point. The third location specified is the same as the second. Since this location is used for curve fitting, the slope of the ending part of the drawn curve is nearly flat.

DOT

The DOT instruction draws one dot on the screen. The syntax is:

```
DOT      address
```

The address can be expressed in any of the three forms of screen addressing.

Remember that each address horizontally specifies one displayable dot, but that there are two addresses vertically for each dot. These odd-even pairs mean that the two following instructions illuminate the same dot:

```
DOT      100,99
DOT      100,100
```

LINE

The LINE instruction draws a straight line between any two points on the screen. The syntax is:

```
LINE     {address;}address
```

The two locations can be specified in any of the three forms of screen addressing. The beginning address is optional. If it is not specified, the line begins at the current location.

The sail, mast, and deck of the sailboat in Figure 10–1 are drawn with the LINE instruction.

VECTOR

The VECTOR instruction draws an arrow. The syntax is:

```
VECTOR   {tail_address;}point_address:arrow_size
```

The first two arguments are the locations of the tail and the point of the arrow. These can be in any of the three forms of screen address. The tail address is optional. If it is not specified, the tail of the arrow is at the current location.

The third argument is a real number between 0 and 1 that determines the length of the two lines for the arrow head. This number specifies a proportion of the total length of the arrow, which determines the size of the arrow head. Figure 10–6 shows arrows of different lengths and with different proportions for the arrow head.

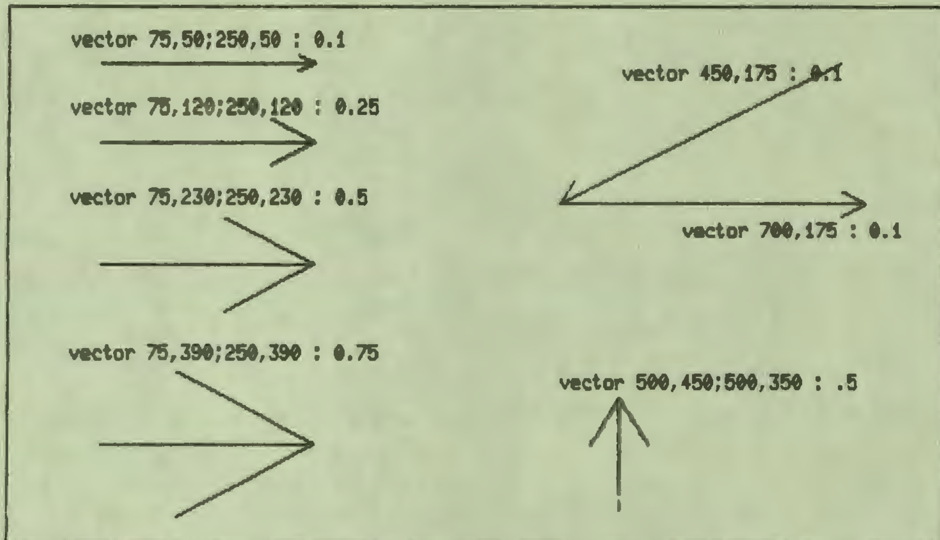


Figure 10–6
The VECTOR Instruction

The four arrows on the left side of Figure 10–6 are the same length, but specify a different proportion of the length for the arrow head.

The two linked arrows on the right show the results of using the current location to specify the tail of the arrow. The following instructions draw and label these two arrows:

```
AT      500,50
WRITE   vector 450,175 : 0.1
VECTOR  450,175 : .1
VECTOR  700,175 : 0.1
AT      550,190
WRITE   vector 700,175 : 0.1
```

The current location after the first line of text is the top left corner of the next character cell. The tail of the diagonal arrow begins at the current location. After the diagonal arrow is drawn, the current location is the point of the arrow. The horizontal arrow begins at the current location.

MODIFYING LINE GRAPHICS

Two instructions modify graphics drawn with the basic graphics instructions. The PATTERN instruction draws lines in different patterns. The SREF instruction shades figures with the foreground color in the current pattern, or with a specified character.

PATTERN

The PATTERN instruction specifies the current pattern for drawing lines. The syntax is:

PATTERN keyword

The keywords are DOT, DASH, DASHDOT, and SOLID. The default pattern is SOLID. After the current pattern is changed, all lines drawn with the basic graphics instructions and the relative graphics instructions are drawn in the specified pattern. Areas shaded with the SREF instruction are shaded in the current pattern.

Figures 10–3, 10–4, and 10–5 show the DASH pattern. Figure 10–7 shows more examples of patterns.

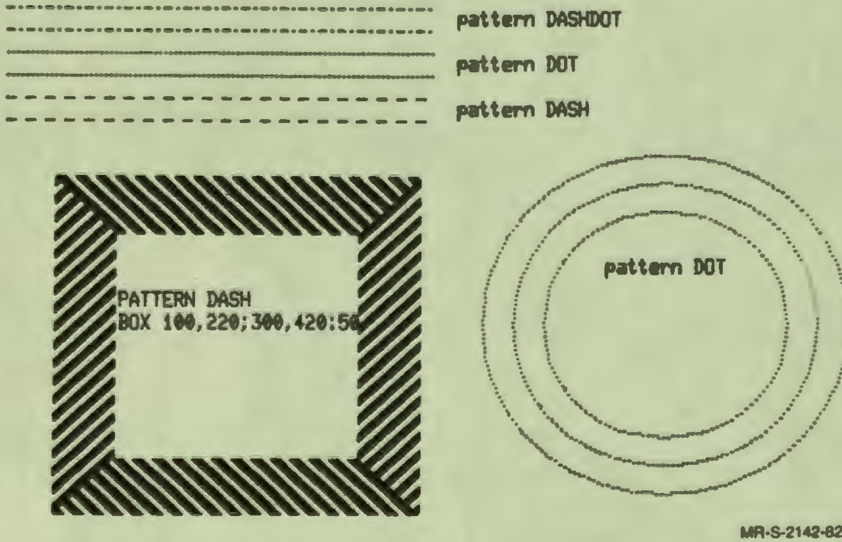


Figure 10–7
The PATTERN Instruction

SREF

The SREF instruction specifies that subsequent graphics should be shaded from the lines they are drawn with to the reference line or point specified by SREF. The current foreground color shades the area between the reference line or point and the lines drawn with the graphics instructions. Shading is in the current pattern, or in a character pattern specified in the SREF instruction. The syntax is:

```
SREF     x_coordinate | NONE, y_coordinate | NONE{, "character"}
```

The first argument, x_coordinate, is the address of a point on the x-axis of the screen expressed as a fine coordinate. Y_coordinate is the address of a point on the y-axis of the screen expressed as a fine coordinate.

The second argument, NONE, is a keyword used in place an x- or y-coordinate in order to create a horizontal or vertical reference line.

The last argument, character, is a single keyboard character, enclosed in quotation marks, to be used as the shading pattern. This argument is optional. If a character pattern is not specified, the current shading pattern is used.

Authors can specify vertical or horizontal reference lines for shading by using the keyword NONE. To establish a vertical reference line, specify an x-coordinate, but enter the keyword NONE in place of a y-coordinate. This establishes a vertical reference line through the specified x-coordinate. The following instruction sets up a vertical reference line for shading through the x-coordinate 100:

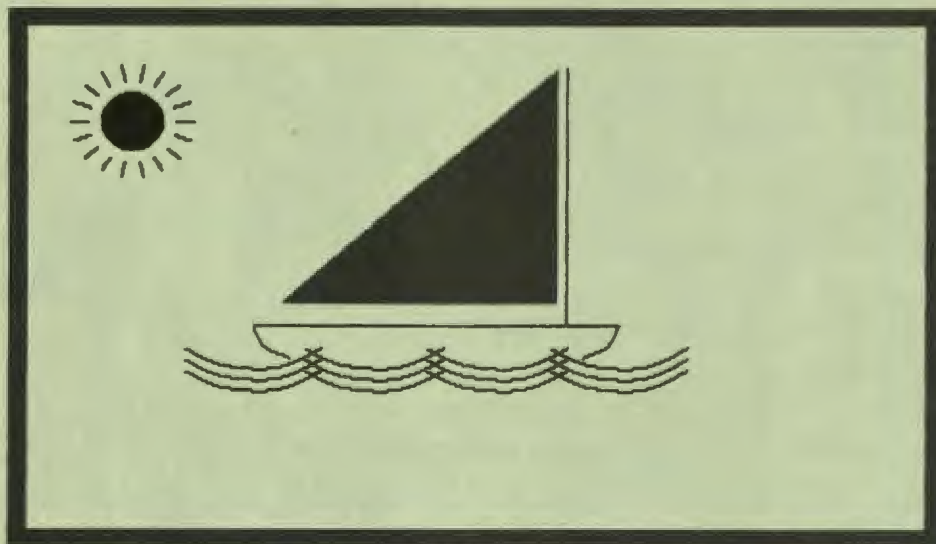
```
SREF     100,NONE
```

To establish a horizontal reference line, enter the keyword NONE in place of an x-coordinate, but specify a y-coordinate. This establishes a horizontal reference line through the y-coordinate.

To establish a shading reference point, specify both the x- and y-coordinates. Shading centers around the reference point and fills out to the lines defined by subsequent graphics instructions.

The SREF instruction with no argument stops shading.

Figure 10–8 shows the sailboat from Figure 10–1 with a shaded sail and a shaded sun.



MR-S-2143-82

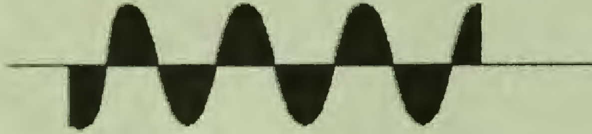
Figure 10–8
Sailboat with Shaded Sail

The following instructions draw the shaded sail:

```
FCOLOR  WHITE
SREF    NONE,260            $$ Set reference line to location
;                           $$ of lower edge of sail.
LINE    448,55;225,260      $$ Draw diagonal line and shade
;                           $$ to horizontal reference line.
SREF                        $$ Turn off shading.
```

After a reference line is specified, each point on any line drawn by any basic or relative graphics instruction is extended vertically to the reference line. If the reference line is above the lines drawn by the instruction, the shading goes up; if the reference line is below, the shading goes down. In the first curve shown in Figure 10–9, the reference line passes through the center of the curve, and the shading goes both up and down. In the second curve, the reference line is below the lowest points of the curve.

```
sref    NONE,150
curve   100,200;150,100;200,200;250,100;300,200;350,100;400,200;450,100;500,200
```



```
sref    NONE,470
curve   50,450;150,350;250,450;350,350;450,450;550,350;650,450;750,350
```

MR-S-3975-85

Figure 10–9
The SREF Instruction

The SREF instruction shades in the current pattern or in a specified character. In the lessons that draw Figures 10–8 and 10–9, the PATTERN instruction is not used, so shading is done in the default pattern SOLID. Figure 10–10 shows examples of shading with the DASH, DASHDOT, and DOT patterns. The instructions that draw each figure are also on the illustration.

Notice that the PATTERN instruction precedes the SREF instruction. When the SREF instruction is executed to begin shading, the current pattern is also selected. Changing the current pattern after an SREF instruction has no effect on the pattern for shading until another SREF instruction is executed.

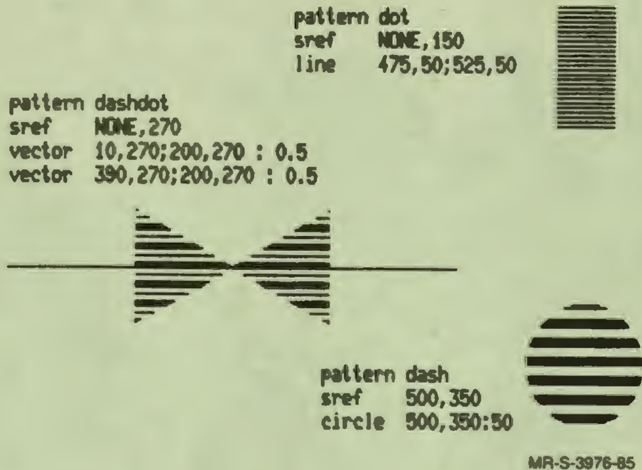


Figure 10-10
The SREF and PATTERN Instructions

TEXT ENHANCEMENT

The SIZE instruction, which changes text size, is explained in Chapter 3. There are two other instructions that modify text: ITALICS and TROTATE. These three instructions can be used together, so that you can display tall, narrow, italicized characters or short, wide, rotated characters.

ITALICS

The ITALICS instruction displays italics. The vertical elements of characters are slanted, while the horizontal elements remain the same. Unlike printed italics, the vertical elements on the screen can be slanted to the left or to the right.

The syntax of the instruction is:

ITALICS degree

Any degree of italics can be specified. The entire text cell is slanted from the top left corner. Positive arguments, such as 45 or 20, move clockwise toward the bottom of the text cell. Negative arguments, such as –30 and –45, move counterclockwise toward the bottom of the text cell.

All text displayed after an ITALICS instruction is italicized until you use the ITALICS instruction with the argument 0 to return text to its normal position.

Figure 10–11 shows different text sizes and degrees of italics.

Figure 10–11
The ITALICS Instruction

TROTATE

The TROTATE instruction displays text with a single text cell rotated around the top left corner. When several characters are displayed by one WRITE instruction, the entire string is rotated.

The syntax of the instruction is:

TROTATE degree

The degree of rotation can be any multiple of 45. If you specify a positive number, the rotation is counterclockwise. If you specify a negative number, the rotation is clockwise.

Subsequent text is rotated until you use the instruction TROTATE 0 to return text to its normal position. When text is rotated, the current location after a string is displayed is the next character position with the same rotation.

Figure 10–12 shows rotated text.

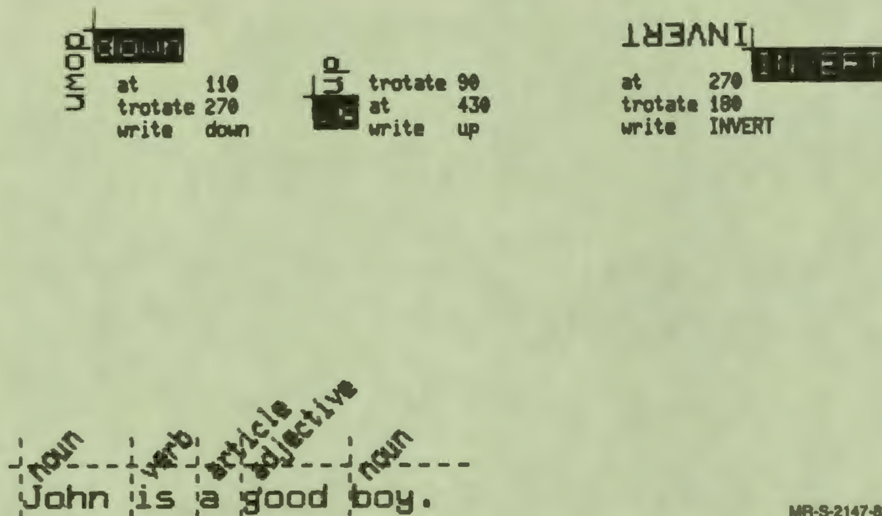


Figure 10–12
The TROTATE Instruction

The top part of Figure 10–12 shows three words in inverse mode and no rotation. Each word is also displayed in normal mode, and rotated a different number of degrees. The instructions for rotating each text string are on the illustration. The cross at the upper left corner of the first character cell of the unrotated strings marks the point around which rotation occurs.

The following instructions draw the sentence marked with the parts of speech for each word.

```
DEFINE    xsize2 = 18               $$ For calculating width
PATTERN   DASH                      $$ of size 2 chars.
LINE      30,380;30,450             $$ Vertical marker line.
LINE      20,406;400,406            $$ Horizontal marker line.
PATTERN   SOLID
TROTATE   0                         $$ Set to no rotation.
SIZE      2
AT        30,420
WRITE     John is a good boy.
TROTATE   45
SIZE      1
AT        30,406
WRITE     noun
LINE      30 + (xsize2*5),380; 30 + (xsize2*5),450
AT        30 + (xsize2*5),406
WRITE     verb
LINE      30 + (xsize2*8),380; 30 + (xsize2*8),450
AT        30 + (xsize2*8),406
WRITE     article
LINE      30 + (xsize2*10),380; 30 + (xsize2*10),450
AT        30 + (xsize2*10),406
WRITE     adjective
LINE      30 + (xsize2*15),380; 30 + (xsize2*15),450
AT        30 + (xsize2*15),406
WRITE     noun
PAUSE
TROTATE   0
ENDLESSON
```

Notice that the horizontal text in this illustration is size 2, while the rotated text is size 1. Text rotated 45, 135, 225, or 315 degrees is also larger.

ERASE

Using the ERASE instruction with no argument erases the entire screen. The ERASE instruction was used with no argument in many of the preceding examples. ERASE can also be used with two arguments to erase any rectangular area on the screen. The syntax is:

```
ERASE     corner;opposite_corner
```

Any of the three forms of screen addressing can be used.

MODE

The MODE instruction modifies the display mode of both text and graphics, and controls features such as blinking, inverse video, overlaying characters, and writing in erase mode.

The syntax of the MODE instruction is:

```
MODE    keyword
```

The three current modes are selected by a keyword from each of the three groups listed below. In each group, the keywords are mutually exclusive. Only one mode in each group is active at any one time. For example, when REPLACE mode is selected with the keyword REPLACE, it remains in effect until a mode instruction with the keyword OVERLAY is used.

- NORMAL/INVERSE

  NORMAL is the default. With text, NORMAL displays the dots in a text cell in the foreground color and the rest of the cell in the background color. With graphics instructions, NORMAL draws lines in the foreground color. INVERSE displays the dots in a text cell in the background color and the rest of the cell in the foreground color. INVERSE draws graphics in the background color. INVERSE should be used with care with graphics. In some instances, graphics are not displayed in INVERSE mode.

- FIXED/BLINK

  FIXED is the default. BLINK alternates between normal video and reverse video. In BLINK mode, text and graphics alternate between the current foreground color and DARK. Note that BLINK mode cannot be used on all terminals.

- OVERLAY/REPLACE/COMPLEMENT/ERASE

  OVERLAY is the default. In a terminal, there is a special display memory called the bit-map that determines which dots are illuminated on the monitor. There is one location in this memory (one bit) for each addressable dot in the x-direction, and one bit for two addressable dots in the y-direction. The WRITE instructions and all graphics instructions change the screen display by setting and clearing bits.

  In OVERLAY mode, the bits for the character cell or for the graphics figure are set. The dots defined by the bits are turned on, and the dots are illuminated.

  In REPLACE mode, the bits affected are cleared before they are set.

  In COMPLEMENT mode, the new display depends on what was previously displayed in the same locations. Each bit is set if it was previously clear, and cleared if it was previously set.

  In ERASE mode, the area defined by a text cell or a graphics instruction is erased instead of being drawn. The bits are all cleared. If something was written in the area, it is erased; otherwise, there is no change.

In addition to using the MODE instruction with keywords, authors can change display modes with one of the following system constants. Using the constant name has the same effect as using the keyword. For example, MODE M_INVERSE has the same effect as MODE INVERSE.

Instruction	Constant
MODE NORMAL	M_Normal
MODE INVERSE	M_Inverse
MODE FIXED	M_Fixed
MODE BLINK	M_Blink
MODE OVERLAY	M_Overlay
MODE REPLACE	M_Replace
MODE COMPLEMENT	M_Complement
MODE ERASE	M_Erase

Figure 10–13 shows the results of OVERLAY, REPLACE, COMPLEMENT, and ERASE modes when they are combined with NORMAL mode. The text on the two sides of the illustration is the same. The box on the left is first shaded using SREF. The same text is then printed on both sides of the illustration to show the results of displaying text in different modes when bits are previously set. Figure 10–14 repeats the same illustration using INVERSE mode.

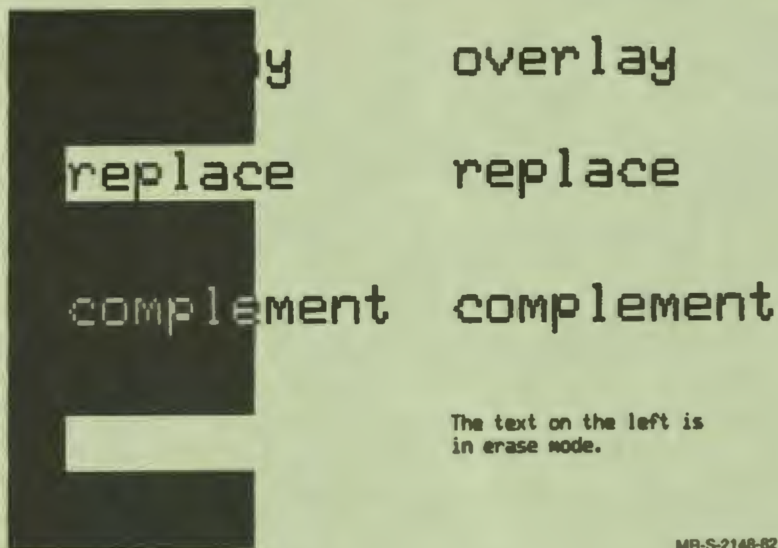


Figure 10–13
The MODE Instruction 1

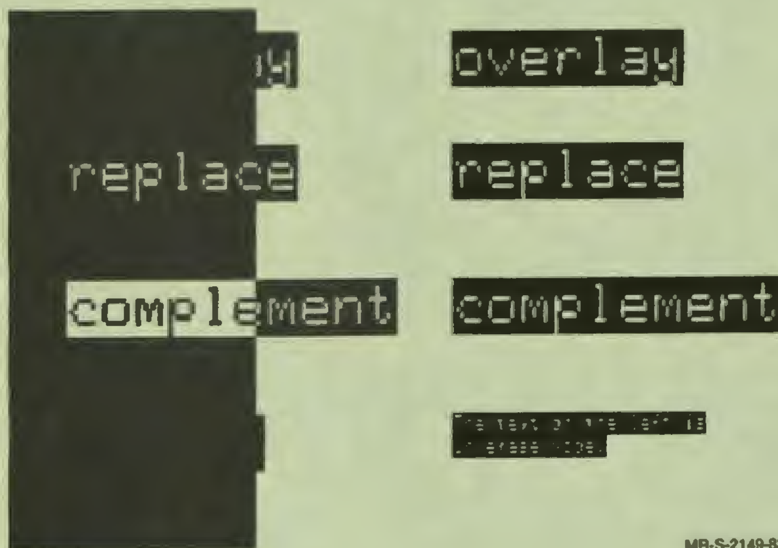


Figure 10–14
The MODE Instruction 2

ERASE mode is useful for erasing only text or graphics previously displayed. The ERASE instruction erases a rectangular area or the whole screen. To erase a circle, use the instruction MODE ERASE; then draw the circle again. Then, reset the display mode to OVERLAY or REPLACE.

COMPLEMENT mode is useful for animation. First select COMPLEMENT mode and draw the figure. In the bit-map memory, the bits for the figure change states, and the figure appears on the screen. Then draw the figure again. The bits again change states, returning to what they were before the figure was displayed. The figure disappears, but nothing else on the screen is changed. The lesson Icecream listed in Appendix F uses COMPLEMENT mode to draw and redraw the drip from the ice cream.

RELATIVE GRAPHICS

In all three coordinate systems, the origin of the coordinates is at the top left corner of the screen. Relative graphics specify any location on the screen as the origin. Then you can use any form of screen addressing, and the locations on the screen are relative to the new 0,0 point.

The following relative graphics instructions perform the same functions and use the same arguments as the basic graphics instructions. The relative graphics instructions, which are listed below, begin with the prefix R.

- RAT
- RBOX
- RCIRCLE
- RCURVE
- RDOT
- RLINE
- RVECTOR.

There are three additional relative graphics instructions:

- RORIGIN specifies the current 0,0 point
- ROTATE rotates the graphics around the origin
- RSIZE modifies the size of relative graphics

You can do some things with relative graphics that you cannot do with the basic graphics instructions. You can write a unit that draws a picture, then display the picture at different locations on the screen by setting the relative origin and executing the unit. Using the basic graphics instructions requires a different series of instructions with different addresses for each display.

Since relative graphics can be scaled, it is easy to display a picture in different sizes, and to draw figures such as ellipses.

Because relative graphics can be rotated, it is easy to draw things like the sun in the sailboat picture.

Figures drawn with the relative graphics instruction are drawn in the current pattern and are shaded with the SREF instruction.

RORIGIN

The RORIGIN instruction selects the screen address to be used as the 0,0 address for subsequent R-prefixed graphics instructions. The syntax is:

```
RORIGIN address
```

Any of the three forms of screen addressing can be used.

Figure 10–15 shows three boxes. The same unit draws all three boxes using the relative graphics instruction RLINE. At lesson level there are three series of instructions. Each RORIGIN sets the current relative graphics origin. Each RAT sets the current location relative to this origin. The address is the same in each case. Each WRITE labels a box, and each DO draw draws one box relative to the current origin.

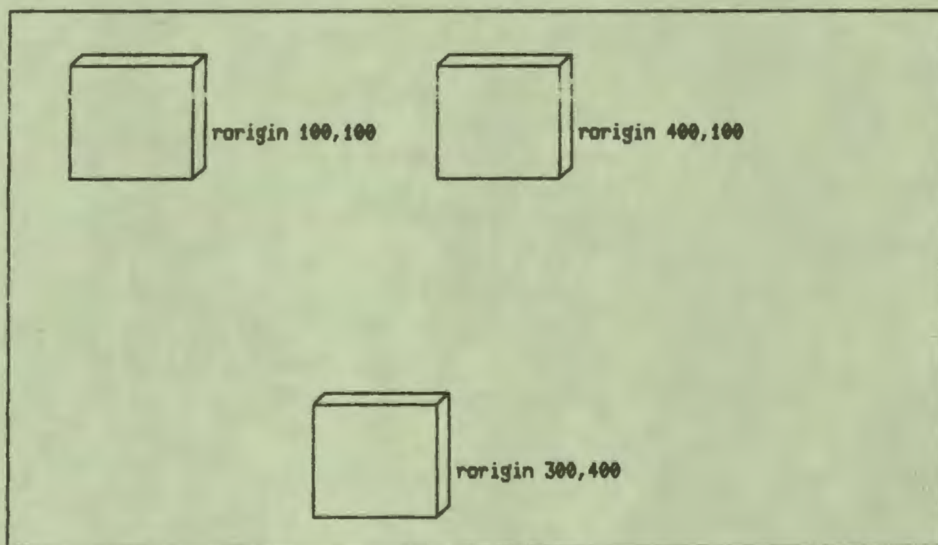


Figure 10–15
The RORIGIN Instruction

The following lesson draws Figure 10–15.

```
LESSON   rotpic
BOX      0,0;0.999,0.999
ERASE
RORIGIN  100,100
RAT      65,0
WRITE    rorigin 100,100
DO       draw
RORIGIN  400,100
RAT      65,0
WRITE    rorigin 400,100
DO       draw
RORIGIN  300,400
RAT      65,0
WRITE    rorigin 300,400
DO       draw
PAUSE
```

```
UNIT      draw
RLINE     -50,-50;-50,50       $$ Left vertical line
RLINE     50,50                $$ Bottom line
RLINE     50,-50               $$ Right vertical line, main box
RLINE     -50,-50              $$ Top line, main box
RLINE     -40,-60              $$ Angle from top left corner
RLINE     60,-60               $$ Top line, perspective box
RLINE     50,-50;60,-60        $$ Angle to top right
RLINE     60,40                $$ Right vertical, perspective
RLINE     50,50                $$ Angle to bottom right
ENDLESSON
```

Notice that many of the RLINE instructions in the unit draw have only one argument. This argument specifies the ending point of the line. The line begins at the current location. In each case, the current location is the ending point of the line drawn by the preceding RLINE instruction.

RSIZE

The RSIZE instruction specifies a size for subsequent R-prefixed graphics instructions. The distances between points in the figure are multiplied by the specified size. Height and width can be changed independently.

The syntax is:

```
RSIZE     x-size,y-size
```

Figure 10–16 shows two boxes of different sizes.

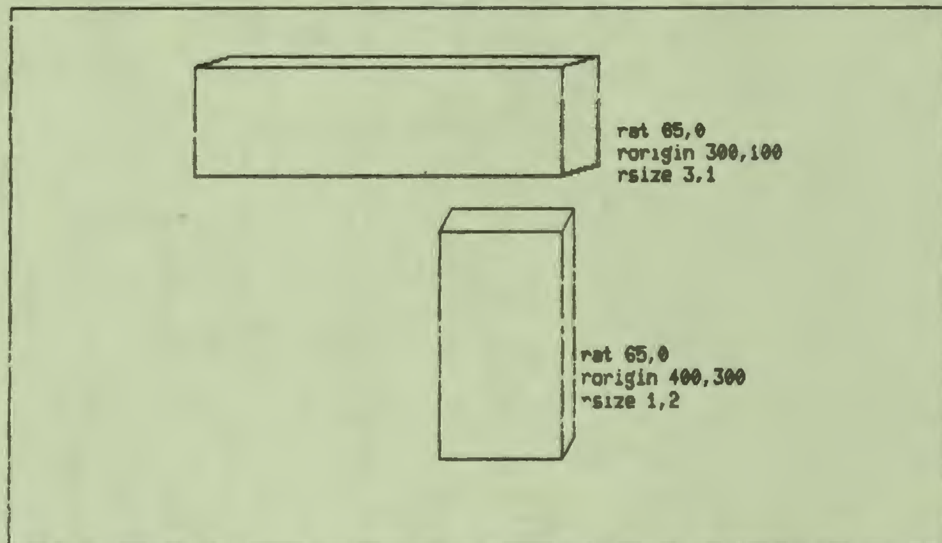


Figure 10–16
The RSIZE Instruction

The boxes in Figure 10–16 are drawn by the same unit as the boxes in Figure 10–15. The RSIZE instruction has changed their proportions. The text written by each box shows the RAT instruction that specified the text location, the RORIGIN instruction that specified the current 0,0 location, and the RSIZE instruction that specified the proportions. RSIZE changes the location specified with RAT just as it changes the locations specified with RBOX and RLINE to draw the figure. It does not change text size.

The following instructions draw Figure 10–16.

```
ERASE
BOX      0,0;0.999,0.999
RORIGIN  300,100      $$ Set origin.
RSIZE    3,1          $$ Set relative size.
RAT      65,0         $$ Label box.
WRITE    rat 65,0
         rorigin 300,100
         rsize        3,1
DO       draw         $$ Draw box.
RORIGIN  400,300
RSIZE    1,2
RAT      65,0
WRITE    rat 65,0
         rorigin 400,300
         rsize 1,2
DO       draw
PAUSE
RSIZE    1,1          $$ Reset relative size for
                      $$ next pictures.
```

ROTATE

The ROTATE instruction rotates the addresses of subsequent relative graphics instructions before the figure is drawn. This makes it possible to draw a figure at an angle without changing the addresses. The syntax is:

```
ROTATE   degree
```

A positive number of degrees rotates the addresses counterclockwise. A negative number rotates the addresses clockwise. The location of each point specified in the instruction is rotated around the origin, then the figure is drawn.

Figure 10–17 shows some lines rotated different numbers of degrees.

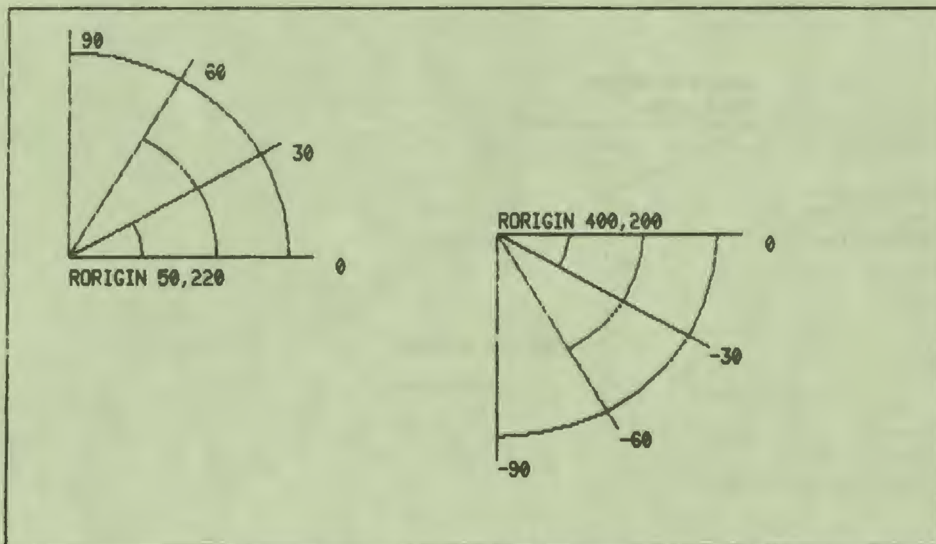


Figure 10–17
The ROTATE Instruction

The sun in the sailboat picture shown earlier is drawn with the following instructions:

```
RORIGIN  100,100    $$ Set relative origin.
SREF     NONE,100   $$ Set reference line at y-coordinate 100.
FCOLOR   YELLOW     $$ for 0 of relative addresses.
RCIRCLE  0,0:25     $$ Draw the sun.
SREF                $$ Stop shading.
DEFINE   c:INTEGER
FOR      c:=0,360,20
.        RLINE      35,0;50,0 $$ Draw one ray.
.        ROTATE c   $$ Rotate addresses.
ENDFOR              $$ Repeat at different angles.
```

Any figure drawn with an R-prefixed graphics instruction can be rotated. The addresses are rotated the number of degrees specified, and the instruction is then executed. With RBOX, this method of rotation produces a rectangle, but the proportions of the rectangle are different at different rotations. When you draw a box with RBOX (and with BOX), you are actually specifying the diagonal of the rectangle. The addresses specifying the diagonal are rotated, and then the rectangle defined by the new diagonal is drawn.

Figure 10–18 shows two boxes drawn with the RBOX instruction. One box is rotated; the other box is not.

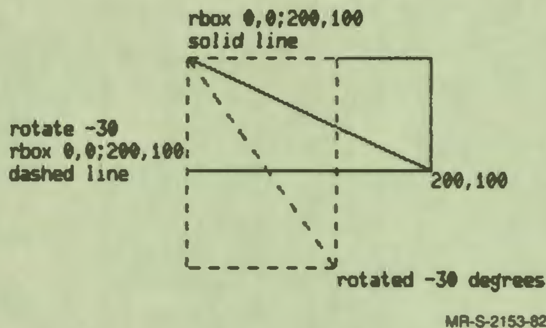


Figure 10–18
Rotated Boxes: RBOX

To rotate a rectangle and have it keep the same proportions, draw the rectangle using four RLINE instructions. Figure 10–19 is another illustration of the same box drawn in Figures 10–14 and 10–15. Because this square is drawn with the RLINE instruction, it remains a square as it is rotated.

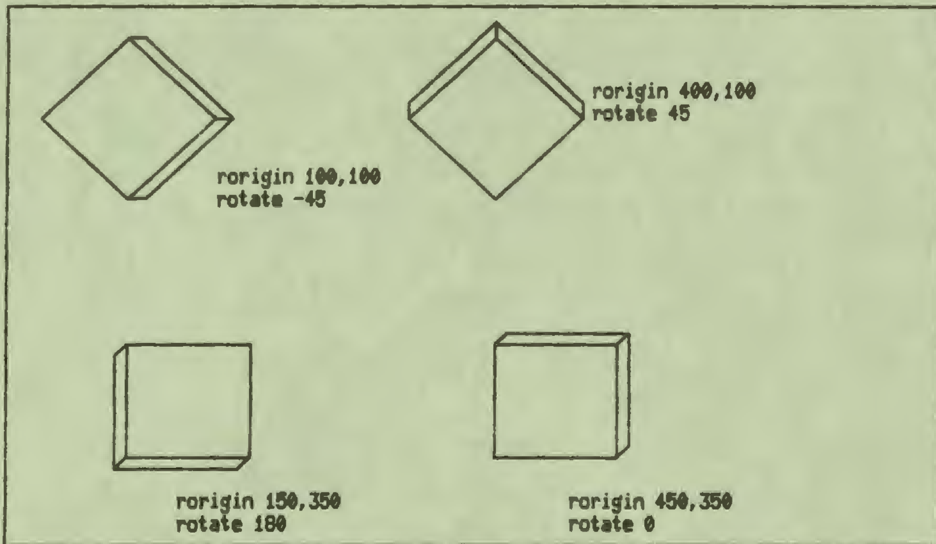


Figure 10–19
Rotated Boxes: RLINE

GRAPHICS SYSTEM VARIABLES

Many of the graphics instructions modify the values of graphics system variables. The system variables provide a convenient way of using or testing the current location, current foreground color, and other current attributes.

Table 10–1 lists some graphics system variables. Uses of these variables are illustrated in the unit that follows the table. Appendix C lists all system variables.

Table 10–1: Graphics System Variables

Name	Data Type	Description	Cause of Update
BCOLOR	integer	Value of the current background color.	BCOLOR instruction
FCOLOR	integer	Value of the current foreground color.	FCOLOR instruction
SIZEX	integer	X size last specified or size when SIZE has only one argument.	SIZE instruction
SIZEY	integer	Y size last specified. A SIZE instruction with only one argument changes this variable to the default y size associated with the argument.	SIZE instruction
WHERE	integer row and column coordinates	Row and column address of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell. After line graphics are displayed, the variable contains the address of the upper left corner of the text cell in which the last dot was drawn	Updated by all graphics instructions. Reset to 000 by ERASE with no arguments.
WHEREX	integer fine coordinates	X-coordinate of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell.	Updated by all graphics instructions. Reset to 0 by ERASE with no arguments.

Table 10–1: Graphics System Variables (Cont.)

Name	Data Type	Description	Cause of Update
WHEREY	integer fine coordinates	Y-coordinate of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell.	Updated by all graphics instructions. Reset to 0 by ERASE with no arguments.

The following unit shows one use of the graphics system variables. Many lessons repeat the same message each time a student must press the RETURN key to continue a lesson. The following unit uses the system variables to save the current location, text size, and foreground color while it displays the press return message, and restores the values before it returns to the calling unit. Another unit in the lesson can call this unit without affecting the display in the calling unit. An alternative way to call another unit without affecting the current display is to use the SAVE and RESTORE instructions, which are explained in Chapter 7.

```
UNIT     press_ret
DEFINE   old_wherex,old_wherey,old_fcolor,old_size : INTEGER
DEFINE   start_mes = 2260
ASSIGN   old_wherex := WHEREX          $$ Save the current values.
ASSIGN   old_wherey := WHEREY
ASSIGN   old_fcolor := FCOLOR
ASSIGN   old_size := SIZEX

SIZE     1                             $$ Display the message.
IF       (BCOLOR = WHITE)              $$ Test value of sys var
.        FCOLOR   BLUE                 $$ BCOLOR, and set FCOLOR
ELSE                                   $$ accordingly.
.        FCOLOR   WHITE
ENDIF
AT       start_mes
WRITE    Press
MODE     INVERSE
AT       WHERE + 1                 $$ Use current location in
                                   $$ sys var WHERE, and skip one
                                   $$ space after Press.
```

```
WRITE   RETURN
MODE    NORMAL
PAUSE
ERASE   WHERE + 100;start_mes    $$ Use sys var WHERE + 100 to
                                 $$ erase message.
AT      old_wherex,old_wherey    $$ Restore the old values.
SIZE    old_size
FCOLOR  old_fcolor
; End of unit.
```

First, the unit press_ret defines local variables for the display attributes it is going to modify and assigns the current values to the variables. Then the unit displays its message, pauses, and erases its message. By using the local variables as arguments, the last three instructions restore the previous values of the system variables.

Any other unit in the lesson can call press_ret. When press_ret returns, the current location, foreground color, and text size are the same as they were before press_ret was executed.

The starting address of the message is defined by the named constant start_mes. With this constant and the system variable WHERE, the message can be displayed anywhere on the screen by changing only the value of the constant. The AT start_mes instruction sets the current location to the address defined by the constant. After the first WRITE instruction has displayed the word Press, the system variable WHERE contains the address of the next character cell in row-and-column coordinates. The argument to the next AT instruction sets the current location one character cell to the right, in effect skipping one space. This is the correct address for writing the word RETURN.

The use of the system variable WHERE and the named constant start_mes with the instruction ERASE is similar. After the word RETURN is displayed, WHERE contains the current location. Adding 100 to this value calculates the address one row below the current address. This is the correct address for the beginning of the rectangle to be erased. Since start_mes contains the address of the beginning of the string, the ERASE instruction erases only the two words.

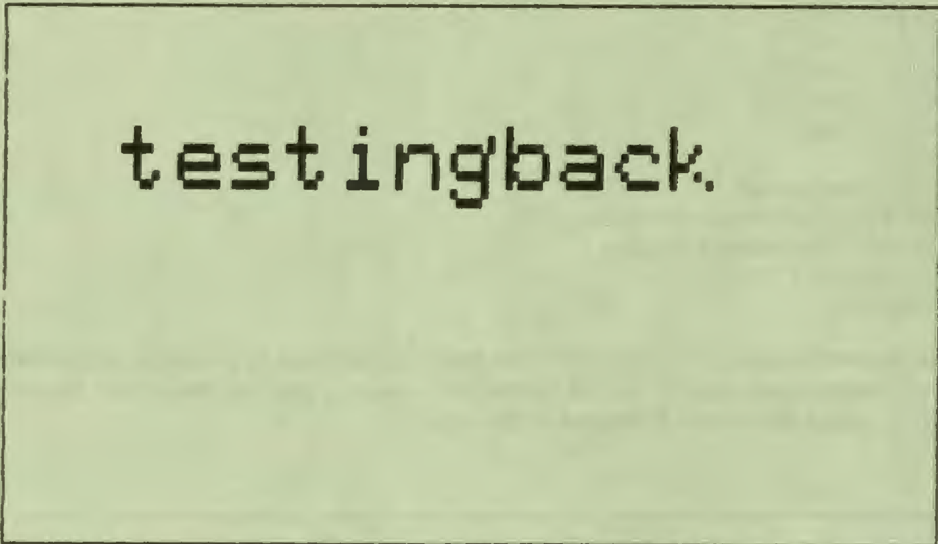
The following lesson displays one word, calls the unit press_ret, then displays another word.

```
LESSON   testing
SIZE     3
FCOLOR   GREEN
AT       510
WRITE    testing
DO       press_ret
WRITE    back
PAUSE
; End of lesson level.
; The instructions shown above are
; inserted here before the lesson
; is compiled.
ENDLESSON
```

The lesson pauses twice, once in the unit press_ret and once at the end of the lesson level instructions. Figure 10–20 shows the screen in the unit press_ret. Figure 10–21 shows the screen at the end of the lesson.

MR-S-2155-82

Figure 10–20
Graphics System Variables 1



MR-S-2156-82

Figure 10–21
Graphics System Variables 2

GENERAL GRAPHICS CONSIDERATIONS

Some of the factors that affect displays are not under the author's control. DAL lessons can be displayed on color monitors or on black-and-white monitors. Terminals can operate at a number of different line speeds. Line speed affects the speed at which text and graphics are displayed.

Different types of terminals often use different processes to generate graphics on their screens. In some cases, observable differences occur when the same lesson executes from two different terminal models. Chapter 7 in this guide discussed these differences, which may be important if your lesson is delivered on more than one type of terminal.

Using Color

When lessons are displayed on a black-and-white monitor, the color keywords or slot numbers select shades of gray. The effects of gray shades and color are different. If you know that your lessons may be displayed on both types of monitors, it is a good idea to test the lessons on both types.

Most monitors have both brightness and contrast controls that can be adjusted by the student. The settings of these controls affect both gray shades and color. Some settings may cause the darker grays or the colors blue and red to be invisible when the background color is DARK. Lessons that begin with text or graphics using these color combinations can appear to be malfunctioning.

On black-and-white monitors, the gray shades nearest to each other, such as white and yellow or magenta and cyan, may not be distinguishable. On color monitors, the colors are distinct. (The colors on different monitors may not be identical. Color monitors, like color television sets, have a number of internal adjustments.)

Terminal Line Speed

The line speed at which terminals operate depends both on the physical connection between the terminal and the computer and on decisions made by the system manager at each site. The line speed affects the speed at which text and graphics are displayed on the screen. Line speeds affect every program, not just DAL lessons.

When terminals are connected to the system over phone lines, 1200 baud is a common line speed. When terminals are connected directly to the computer, 4800 baud and 9600 baud are common line speeds. The difference between 1200 baud and 4800 baud is noticeable, but the difference between 4800 baud and 9600 baud is usually not apparent.

Animation, which in general requires displaying a figure and then erasing it at a number of different locations, looks different at 1200 baud than at 4800 or 9600 baud. If you know that your lessons can be run on terminals with different line speeds, test the lessons at different speeds.

Displaying text is faster than displaying graphics. Lessons that display primarily text and use only simple graphics such as underlining words generally look right at 1200 baud.

Line speed is rarely a problem for student input from the keyboard.

11

Modifying Lesson Flow

11

Modifying Lesson Flow

LOOP/ENDLOOP, FOR/ENDFOR, and other control logic instructions were explained in Chapter 5. This chapter discusses several other VAX DAL control logic instructions that can modify lesson flow.

Use VAX DAL control logic instructions to do the following:

- Direct lesson flow back to lesson level
- Direct lesson flow to another unit in the current unit calling chain
- Respond to specific strings in student responses
- Execute a unit at regular time intervals or at a specified time of day
- Respond to specific instructions in the lesson
- Respond to technical problems that arise during lesson execution

Topics discussed in this chapter include:

- The unit calling chain
- Unconditional transfer of control
  - RETURN
  - BACKUP
- Conditional transfer of control
  - BRANCH
  - WHEN
  - Condition handling

A transfer of control from unit level to lesson level, or from one unit to another, can only occur along the unit calling chain. This chapter begins with a discussion of the unit calling chain.

THE UNIT CALLING CHAIN

A unit's calling chain is its lineage. In the same way that a family lineage traces a path from an individual back to an ancestor, the calling chain traces a path from the current unit back to lesson level.

To extend the comparison, units, like individuals, can be of different generations. A first-generation unit is a unit that is executed by a DO instruction at lesson level. A second-generation unit is executed by a DO instruction in a first-generation unit. A third-generation unit is executed by a DO instruction in a second-generation unit, and so on.

The figure below illustrates the calling chain of a very simple lesson, lesson Call_Chain. This lesson executes three units in a single calling chain. Lessons can contain any number of calling chains, which execute one at a time.

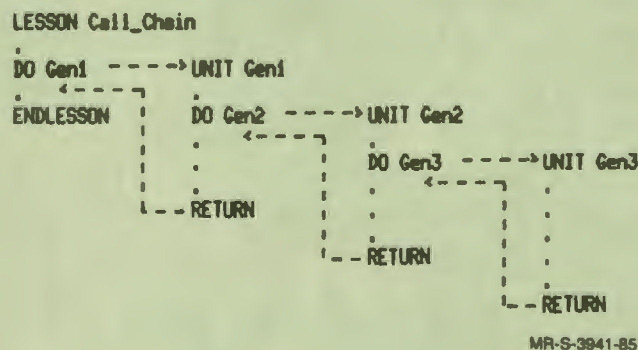


Figure 11–1
A Unit Calling Chain

Although control passes from unit Gen1 to unit Gen2 at the DO instruction, unit Gen1 remains open in the calling chain while Gen2 and Gen3 execute. Unit Gen1 remains open until control is returned to Gen1 and execution of Gen1 finishes.

A unit's place in the calling chain has a direct bearing on what can and cannot be done with control logic instructions in the unit, as the next two sections in this chapter explain.

UNCONDITIONAL TRANSFER OF CONTROL

VAX DAL provides two instructions that immediately redirect lesson flow along the unit calling chain. These are the RETURN and BACKUP instructions.

The RETURN Instruction

Whenever a RETURN instruction is encountered in a unit, execution of the unit is immediately terminated. Control is then transferred to the instruction following the last instruction executed in the unit or level that called the current unit. RETURN transfers control back to the unit that executed the current unit, or, if the unit is a first-generation unit, transfers control back to lesson level. Lesson flow is redirected one level back up the calling chain.

The syntax of the RETURN instruction is:

```
RETURN (RET)
```

RETURN can be used as a response-contingent instruction.

The BACKUP Instruction

Like the RETURN instruction, BACKUP redirects lesson flow to another level in the current unit calling chain. Unlike the RETURN instruction, BACKUP can transfer control two, three, or more steps along the calling chain.

The BACKUP instruction has three options:

- BACKUP TOP
- BACKUP unit_name
- BACKUP number

BACKUP TOP terminates execution of the unit and passes control to the instruction following the last instruction executed at lesson level.

BACKUP unit_name passes control to a unit specified by unit_name. Enclose the unit_name in double quotes. The specified unit can only be a a unit executed earlier in the current unit calling chain — that is, the unit must have either directly or indirectly called the current unit. Note that control can be passed only to DAL units, and not to DAL functions, subroutines written in other programming languages, or condition units (discussed later in this chapter). A BACKUP unit_name instruction is ignored if it specifies a unit that:

- does not exist
- is not part of the current unit calling chain

BACKUP number passes control the specified number of units back up the calling chain. For example, a BACKUP 3 instruction in a fourth-generation unit returns control to the first-generation unit in the chain. The lesson terminates if BACKUP number specifies a transfer of control beyond lesson level (for example, specifies a return six generations up the calling chain from a second-generation unit).

With both BACKUP unit_name and BACKUP number, control is passed to the instruction following the last instruction executed in the level to which control has returned.

The BACKUP instruction works properly only in lessons written entirely in VAX DAL. The calling chain of a unit that executes a BACKUP instruction cannot contain a routine written in another programming language. Otherwise, BACKUP may return control to the wrong lesson location.

Lesson backup displays applications of the BACKUP instruction in all three of its forms.

```
LESSON   backup
DEFINE   review : BOOLEAN  $$ Flag for review
DEFINE   set     : INTEGER  $$ Problem set number
DEFINE   pnum   : INTEGER  $$ Problem number
ASSIGN   set := 1           $$ Start with SET1
ASSIGN   pnum:= 1

FCOLOR   BLUE
DO       table

UNIT     table              $$ A first-generation unit
ERASE
AT       210
WRITE    This lesson tests your multiplication skills.
         All of the problems that you will be asked come
         from the following multiplication table.
$begin_a
SIZE     2
IF       review
.        ERASE
ENDIF    review := FALSE
ASSIGN

AT       610
WRITE    MULTIPLICATION TABLE
              1 X 12 = 12
              2 X 12 = 24
              3 X 12 = 36
              4 X 12 = 48
              5 X 12 = 60
              6 X 12 = 72
SIZE     1
AT       2005
WRITC    Press RETURN when you are ready to do problems.
PAUSE
;
DO       set1

BRANCH   review,$begin_a    $$ Does the student want see the table?

UNIT     set1               $$ A second-generation unit.
DEFINE   i : integer

$begin_b

BRANCH   set=2,$skip1       $$ Go directly to SET2
```

```
ASSIGN   review := FALSE
PROMPT   " = "
ERASE
AT       220
SIZE     2
WRITE    PROBLEM SET-1
SIZE     1

FOR      i := pnum,3
.        ERASE    810;925
.        AT       810
.        WRITE    <<s,i>> X 12
.        QUERY    *
.        RIGHTV   i*12
.        .        AT       1220
.        .        WRITE    Very Good!      -Press RETURN-
.        .        PAUSE
.        .        ERASE    1220;1360
.        WRONG
.        .        AT       1015
.        .        WRITE    No, you're wrong. Would you like to see the
                           multiplication table again?
.        .        PROMPT "(Y/N) >"
.        .        INPUT
.        .        IF       (RESPONSE = "Y") OR (RESPONSE = "y")
.        .        .        ASSIGN   review := TRUE
.        .        .        ASSIGN   pnum := i         $$ Save current problem
.        .        .        BACKUP   1                 $$ To unit TABLE
.        .        ELSE
.        .        .        PROMPT " = "
.        .        .        ERASE    1015;1365
.        .        ENDIF
         ENDQ
ENDFOR

ASSIGN   set := 2                  $$ Now next set.
ASSIGN   pnum:= 4

$skip1
DO       set2

BRANCH review,$begin_b

UNIT     set2                      $$ A third-generation unit.

DEFINE   i:INTEGER
```

```
ERASE
AT          220
SIZE        2
WRITE       PROBLEM SET-2
SIZE        1
PROMPT      " = "

FOR         i := pnum,6
.           ERASE       810;925 .  AT        810
.           WRITE       <<s,i>> X 12
.           QUERY       *
.           RIGHTV      i*12
.           .           AT          1220
.           .           WRITE       Very Good! - Press RETURN
                                    -
.           .           PAUSE
.           .           ERASE       1220;1360
.           WRONG
.           .           AT          1015
.           .           WRITE       No, you're wrong. Would you like to see the
                                    multiplication table again?
.           .           PROMPT      "(Y/N) >"
.           .           INPUT
.           .           IF          (RESPONSE = "Y") OR (RESPONSE = "y")
.           .           .           ASSIGN      review := TRUE
.           .           .           ASSIGN      pnum := i         $$ Save current problem.
.           .           .           BACKUP      "table" $$ Transfers control to lesson level.
.           .           ELSE
.           .           .           WRITC       In that case, would you like to redo
                                                the problem set you just finished?
.           .           .           INPUT
.           .           .           IF          (RESPONSE = "Y") OR (RESPONSE = "y")
.           .           .           .           ASSIGN          review := TRUE
.           .           .           .           ASSIGN          set := 1
.           .           .           .           ASSIGN          pnum:= 1
.           .           .           .           BACKUP          "set1" $$ To the SET 1
.           .           .           ENDIF
.           .           ENDIF
.           .           PROMPT " = "
.           .           ERASE       1015;1665
.           ENDQ
ENDFOR

ENDLESSON
```

CONDITIONAL TRANSFER OF CONTROL

The VAX DAL instructions that transfer control conditionally are the BRANCH, WHEN, and condition-handling instructions.

The BRANCH Instruction

Like the control logic structures discussed in Chapter 5, the BRANCH instruction can only be used in a single level of the lesson. BRANCH does not transfer control to another level or unit. However, the BRANCH instruction does conditionally or unconditionally transfer control in the level in which the instruction is executed. The syntax of the BRANCH instruction is:

BRANCH (TAB)/(SP) {condition,} $label1 {,$label2}

Where:

{condition} is a Boolean expression. This argument is optional.

$label1 is the labeled location to which control is passed if the condition is true, or if a condition is not specified. Labels must begin with a dollar sign ($), and can contain no more than 10 characters.

{$label2} is a second labeled location to which control is passed if the condition is false. This argument is optional.

Whenever a BRANCH instruction without a conditional argument is encountered in a unit or at lesson level, control is passed immediately to the first labeled location specified by the instruction.

If BRANCH specifies a Boolean condition, DAL tests the value of the condition. If the condition evaluates to true:

- DAL transfers control to the first labeled location specified by the BRANCH instruction.

If the condition evaluates to false, DAL does one of two things:

- If the condition is false and BRANCH specifies a second label, DAL transfers control to the location specified by the second label.
- If the condition is false and BRANCH does not specify a second label, the BRANCH instruction is ignored, and control passes to the instruction following BRANCH.

The following example shows an application of the BRANCH instruction at lesson level. Note that this example shows only a partial lesson. A full lesson includes the units called from lesson level.

```
LESSON   stop
DEFINE   review:BOOLEAN,RESTART       $$ Set to TRUE in unit askend.
BRANCH   GOAL = 2,$goal2              $$ If lesson begins with GOAL set to 2
BRANCH   GOAL = 3,$goal3              $$ or 3, branch to appropriate place.
DO       unit1
GOAL     GOAL+1                       $$ Increment goal for next time.
SCORE    FALSE                        $$ Don't score query in unit askend.
DO       askend                       $$ Ask student about ending.
SCORE    TRUE                         $$ Turn on scoring and continue.
$goal2
IF       review                       $$ Review is TRUE if student asked
.        .        DO        review1   $$ to stop.
.        ASSIGN      review:=FALSE    $$ Reset for next time.
ENDIF

DO       unit2
GOAL     GOAL+1
SCORE    FALSE
DO       askend
SCORE    TRUE
$goal3
IF       review
.        DO       review2
.        ASSIGN   review:=FALSE
ENDIF
DO       unit3
DO       end
.
.
.
ENDLESSON
```

The above example contains the lesson level instructions for a lesson that is to be started more than once. Students can stop the lesson and then restart the lesson later. Each time they start the lesson, the GOAL system variable is incremented. Because GOAL is saved as a restart variable if the lesson is stopped, GOAL contains the value 2 when the lesson executes a second time. The BRANCH instruction tests to see whether the student is taking the lesson a second (or greater) time. If so, BRANCH transfers control to a location in the lesson different from the location the student started from the first time. In this way, BRANCH creates a lesson that starts with new material the first three times the lesson is executed. (To see the complete lesson, refer to the definition for the STOP instruction in the *VAX DAL Reference Manual*.)

The WHEN Instruction

VAX DAL provides authors with two ways to create lessons that respond to conditions. One way is with WHEN statements. The other way is with condition handlers. WHEN statements are used to execute a unit:

- At a specified time of day
- After a specified period of time
- Each time a specified time interval elapses
- When the student enters a specified string of characters

The syntax of the WHEN instruction is:

WHEN (TAB)/(SP) keyword, condition, {unit_name}

Where:

keyword identifies the type of condition the WHEN statement responds to. The WHEN keywords are listed in the two tables below. Keywords from the first table specify WHEN statements that execute after a specified period of time, at a specified time of day, or every time a regular period of time expires. Keywords from the second table establish, cancel, and restore WHEN statements that respond to specific character strings in student responses.

condition is an expression consistent with the condition type specified by the keyword.

{*unit_name*} is the name of the unit to be executed if the specified condition occurs. If a unit_name argument is not specified, the WHEN instruction cancels a previously declared WHEN instruction that uses the specified keyword and condition.

Table 11–1: Time-Related WHEN Keywords

Keyword	Condition
ELAPSED	The time in seconds which is to elapse between execution of the WHEN instruction and execution of the unit the WHEN instruction specifies. After the specified number of seconds is up, the current unit is interrupted, and control passes to the specified unit.
TIME	The time of day (based on the operating system clock) at which the specified unit is to be executed. When the specified time is reached, the current unit is interrupted, and control passes to the unit specified by the WHEN instruction.
INTERVAL	The time in seconds which is to elapse between executions of the unit specified by the WHEN instruction. Each time the interval is up, the unit executes.

Table 11–2: String-Related WHEN Keywords

Keyword	Action
STRING	The WHEN instruction responds to a string of characters, or to an expression that evaluates to a string. The argument to a WHEN STRING instruction is a string enclosed in double quotes. After the student types the last character of the string, the current unit is interrupted, and control passes to the unit specified by the WHEN instruction.
DISABLE	This keyword cancels a previously declared WHEN STRING instruction. A WHEN STRING instruction lists a string of characters (enclosed in double quotes) as its argument. A WHEN DISABLE "string" instruction cancels a WHEN STRING "string" unit_name instruction if the two string arguments match. A WHEN DISABLE instruction does not specify a unit_name.
ENABLE	This keyword restores a previously canceled WHEN STRING instruction. A WHEN ENABLE "string" instruction restores a canceled WHEN STRING "string" instruction if the two string arguments match.

The unit specified by a WHEN instruction executes anytime the interrupt condition occurs.

After the unit specified by WHEN is executed, the keyword determines the lesson location to which control is returned. If the keyword is a time-related keyword from Table 11–1, control is returned to the instruction that follows the instruction that was executing when the time interval was satisfied. If the keyword is STRING, control is returned to the beginning of the unit or lesson that was interrupted when the student typed the string.

The scope of a WHEN instruction depends upon the keyword used, and the location of the instruction in the unit calling chain. Regardless of keyword, a WHEN instruction executed at lesson level is in effect for the rest of the lesson.

WHEN instructions that use the time-related keywords from Table 11–1 are global in nature. Once declared, they are in effect for the rest of the lesson. If executed in a unit, a time-related WHEN instruction is still in effect after execution of the unit finishes.

WHEN instructions that use the string-related keywords from Table 11–2 are local to the level in which they are declared. If executed in a unit, a string-related WHEN instruction remains in effect only as long as the unit remains active. As soon as execution of the unit finishes, the WHEN instruction declared in the unit is canceled.

Authors can be selective about where they want a WHEN STRING instruction to apply. A WHEN STRING instruction executed in a first-generation unit applies only to subsequent units in that unit's calling chain. As soon as that first-generation unit finishes execution and returns control to lesson level, the WHEN STRING instruction established by the unit is canceled.

Any WHEN instruction can be canceled. A WHEN instruction that specifies a keyword and a condition, but not a unit_name, cancels a previously declared WHEN instruction that uses the specified keyword and condition. For example, WHEN INTERVAL,60 cancels a previous WHEN INTERVAL,60,unit_name instruction.

A specific WHEN STRING, "string", unit_name instruction is temporarily suspended by a WHEN DISABLE, "string" instruction if the two string arguments match. When execution of the unit containing the WHEN DISABLE instruction finishes, the suspended WHEN STRING instruction is automatically restored. WHEN STRING instructions suspended at lesson level are turned off for the rest of the lesson (unless they are restored by a WHEN ENABLE instruction).

A WHEN STRING instruction can be canceled by a WHEN DISABLE instruction for the duration of a calling chain (or for just part of that calling chain). DAL automatically restores the WHEN STRING instruction when execution of the unit containing the WHEN DISABLE instruction finishes.

The WHEN ENABLE, "string" instruction restores a previously disabled WHEN STRING, "string" instruction if the two string arguments match. WHEN ENABLE can only restore a WHEN instruction that was disabled in the current unit. A WHEN STRING instruction disabled at lesson level can only be restored by a WHEN ENABLE instruction at lesson level.

The DAL code below shows one application of the WHEN STRING instruction. Because the string to which WHEN STRING responds is defined by the author, the string can be anything the author desires: "help", "review", or "stop", for example. The lesson below uses WHEN STRING instructions with the SET FKEY instruction (explained in Chapter 7) to modify the example used earlier in this chapter for the BACKUP instruction.

```
LESSON    helpkey
DEFINE    I : integer
FCOLOR    WHITE
AT        410
SET       FKEY TERMINATE
WHEN      STRING, "[F10_KEY]", table
WHEN      STRING, "[F12_KEY]", probset1
%MACRO    press_ret
AT        2245
MODE      INVERSE
WRITE     ... Press RETURN to continue ...
MODE      NORMAL
%ENDMACRO
WRITE     This lesson tests your multiplication skills.
          Study the following multiplication table because
          all of the problems you will be asked come from it.
          Press RETURN to see the multiplication table.
PAUSE
DO        table
DO        probset1
DO        probset2

WRITC     Press RETURN to end this lesson.
PAUSE

UNIT      table
ERASE
FCOLOR    RED
SIZE      2
AT        525
```

```
WRITE     1 X 12 = 12
          2 X 12 = 24
          3 X 12 = 36
          4 X 12 = 48
          5 X 12 = 60
          6 X 12 = 72
SIZE      1
AT        2010
FCOLOR    WHITE
WRITE     Press RETURN when you are ready to try some problems.
PAUSE
RETURN

UNIT      Probset1
FOR       I := 1,3
.         ERASE
.         AT        310
.         FCOLOR    WHITE
.         WRITE     Problem Set #1
.         PROMPT    "= "
.         AT        810
.         FCOLOR    RED
.         WRITE     <<s,I>> X 12
.         QUERY     *
.         RIGHTV    I*12
.         .         ERASE     1220,2379
.         .         AT        1220
.         .         FCOLOR    YELLOW
.         .         WRITE     Very Good!
.         .         press_ret
.         .         PAUSE
.         WRONG
.         .         AT        1220
.         .         WRITE     No, you're wrong. Press the F10 key
                              if you'd like to see the multiplication
                              table again.
.         ENDQ
ENDFOR
```

```
RETURN
;
UNIT      Probset2
ERASE
FOR       I := 4,6
.         ERASE
.         AT        310
.         FCOLOR    WHITE
.         WRITE     Problem Set #2
.         AT        810
.         FCOLOR    RED
.         WRITE     <<s,I>> X 12
.         QUERY     *
.         RIGHTV    I*12
.         .         ERASE     1220,2379
.         .         AT        1220
.         .         FCOLOR    YELLOW
.         .         WRITE     Very Good!
.         .         press_ret
.         .         PAUSE
.         WRONG
.         .         AT        1220
.         .         WRITE     No, you're wrong. Press the F10 key if
                              you'd like to see the multiplication
                              table again, or the F12 key if you'd like to
                              redo the problem set you just finished.
          ENDQ
ENDFOR
RETURN
ENDLESSON
```

Condition Handlers

Like WHEN instructions, condition handlers respond to conditions that arise during lesson execution. Unlike WHEN instructions, condition handlers respond to:

- Specific instructions in the lesson such as STOP, ENDLESSON, or SIGNAL
- Keys pressed by the student such as CTRL/C
- Technical problems with a DECtalk unit or with file input/output operations

Four different instructions are used with condition handling:

ON	declares a condition handler
CDUNIT	names a unit that is executed by a condition handler
SIGNAL	invokes a condition handler
CANCEL	disables a condition handler declared in the current level of the lesson

The ON Instruction

An ON instruction specifies:

- The condition a condition handler responds to
- The *condition unit* that executes if the specified condition becomes true

If a condition named in an ON instruction occurs, the condition handler interrupts the current unit and executes the condition unit. After execution of the condition unit finishes, control is returned to the instruction that follows the instruction that was executing when the interrupt occurred.

The syntax of the ON instruction is:

```
ON (TAB)/(SP) condition_name, cdunit_name
```

Where:

condition_name is one of the condition names listed in Table 11–3.

cdunit_name specifies the name of a condition unit, which is a DAL unit that begins with a CDUNIT instruction.

Specify the condition a condition handler responds to with one of the following condition names:

Table 11–3: Condition-Name Keywords

Condition Name	Meaning
ANYCONDITION	establishes a default condition handler. A default condition handler responds to any VAX/VMS error condition for which a handler is not already set up. Should such an error occur, the ONCODE system variable is set with the VAX/VMS error code that identifies the error type. Consult the VAX/VMS documentation set for listings of the VAX/VMS error codes.
CONDITION,value	establishes a condition handler that can be deliberately invoked by the lesson. *Value* is an integer that labels the condition handler for a SIGNAL instruction.
CTRL_C	establishes a condition handler set up to execute before DAL's built-in CTRL/C handler. Authors can use this handler to execute a condition unit that restores terminal states if a student aborts the lesson with CTRL/C.
DTSTATUS	establishes a condition handler set up to handle problems with a DECtalk unit. A DTSTATUS condition handler executes anytime the value of the DTSTATUS system variable is not 1. (A DTSTATUS value of 1 indicates a successful DECtalk operation.)
IORESULT,value	establishes a condition handler set up to respond to specific file input/output problems. *Value* specifies the value of the IORESULT system variable that causes the handler to execute.
ENDLESSON	establishes a condition handler that executes at the end of the lesson. Unlike a CTRL_C condition handler, which executes only if the student aborts the lesson, this condition handler executes when the student finishes the lesson.
STOPLESSON	establishes a condition handler that executes at lesson end if the author concludes the lesson session with a STOP instruction.

Of the condition names described above, *CONDITION, value*, and *IORESULT, value* are the only two that require arguments. The other condition names establish handlers that react automatically to instructions encountered in the lesson, keys pressed by the user, or operation status codes reported by VAX/VMS.

The CONDITION, value condition name establishes a condition handler that authors can deliberately invoke. The *value* argument is an integer. A SIGNAL CONDITION, value instruction invokes a CONDITION, value condition handler if the two integer values match. See below for further discussion of the SIGNAL instruction and the CONDITION, value condition handler.

The IORESULT, value condition name establishes a condition handler that executes if file input/output problems set the IORESULT system variable with an anticipated status code. IORESULT has 16 possible values, which are listed in Table 11–4.

Table 11–4: IORESULT values

Value	Meaning
1	normal return
2	invalid channel number
3	insufficient virtual memory space
4	channel not open
5	file is read only
6	file is sequential only
7	file is write only
8	new file was created with OPEN instruction
9	tray not opened (SLIDE expects a previously opened tray file)
10	RMS error; check the RMSSTATUS system variable for the error type
11	file wrong type for operation
12	file not indexed, index operation requested
13	problem with key on FIND or GET instruction
14	restart file not found
15	restart variable missing
16	permanent variable missing

An IORESULT condition handler keyed to one of the above values responds if the value is returned after a file I/O operation.

The scope of a condition handler depends on its place in the calling chain. An ON instruction executed at lesson level creates a condition handler that is in effect for the rest of the lesson. A condition handler declared in a unit is canceled when execution of the unit finishes.

Authors can be selective about where they want a condition handler to apply. An ON instruction in a first-generation unit applies only to subsequent units in the calling chain of that unit. As soon as the first-generation unit finishes execution and returns control to lesson level, the condition handler it established is canceled.

The CDUNIT Instruction

CDUNIT identifies a DAL unit that is associated with a condition handler. Like the UNIT instruction, the CDUNIT instruction begins a series of DAL instructions that are executed together as a unit. The syntax of the CDUNIT instruction is:

CDUNIT (TAB)/(SP) cdunit_name

Where:

> *cdunit_name* is a unique name that identifies the condition unit. Condition unit names contain from one to 10 characters, the first of which must be a letter.

Condition units function like other DAL units except for the following differences.

- System variables that normally clear when a new unit begins execution remain unchanged when a condition unit begins execution.
- Response-related variables and variables that DAL normally saves when a unit is called from within a QUERY block are not saved when a condition unit is called from within a QUERY block.
- Condition units cannot contain QUERY blocks.
- Condition units cannot contain ON instructions
- Condition units cannot contain WHEN instructions

Condition units can be used anywhere in a lesson. For example, condition units can be executed by WHEN or DO instructions. However, authors who use condition units for purposes other than condition handling do so at their own risk. The differences listed above can produce unanticipated results.

The SIGNAL Instruction

The SIGNAL instruction invokes a CONDITION, value condition handler. As explained above, a SIGNAL CONDITION, value instruction invokes a CONDITION, value condition handler if the two values match. For example, A SIGNAL CONDITION,3 instruction immediately transfers control to the condition unit named in an ON CONDITION,3 cdunit_name instruction.

A SIGNAL instruction can only invoke an active condition handler. A condition handler declared in a unit becomes inactive when execution of the unit finishes. A condition handler declared at lesson level is active throughout the lesson, and can be invoked by a SIGNAL instruction at any level of the lesson.

The CANCEL Instruction

The CANCEL condition_name instruction disables a condition handler associated with the specified condition name. For example, the instruction CANCEL IORESULT, 15 disables the condition handler set up to respond to an IORESULT value of 15. The syntax of the CANCEL instruction is:

CANCEL (TAB)/(SP) condition_name

Where:

condition_name is:

ANYCONDITION
CONDITION, value
CTRL_C
DTSTATUS
IORESULT,value
ENDLESSON
STOPLESSON

CANCEL can only disable a condition handler declared at the level of the lesson that is currently executing. A condition handler declared at lesson level can only be disabled by a CANCEL instruction executed at lesson level. A condition handler declared in a unit can only be disabled by a CANCEL instruction in that unit.

A canceled condition handler is disabled for the rest of the lesson (unless the handler is restored with another ON instruction).

A CANCEL instruction has no effect if it specifies:

- A condition_name for which there is no condition handler
- A condition handler that was not declared in the current level of the lesson

A Condition-Handling Example

Lesson handler (shown below) displays the effect of five of the condition handlers described above, as well as the use of the ON, SIGNAL, CANCEL, and CDUNIT instructions.

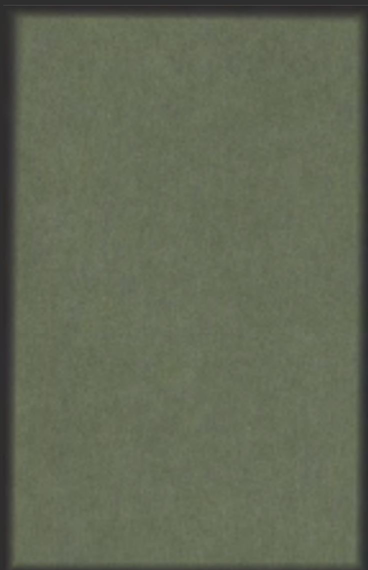
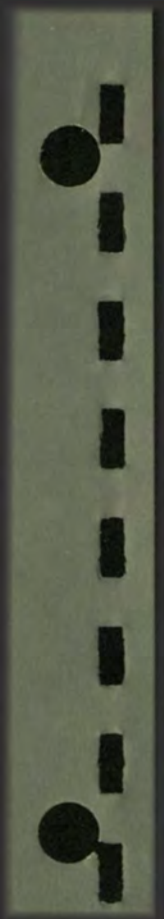
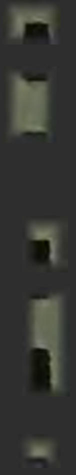
```
LESSON  handler
;
;       Declare   five condition handlers using the syntax:
;                 ON        condition_name, cdunit_name
;
ON      CONDITION,1,cond1
ON      CONDITION,2,cond2
ON      CTRL_C,ctrlc
ON      STOPLESSON,stopl
ON      ENDLESSON,endl
;
;       Display a menu of options.
;
LOOP    true
.       ERASE
.       WRITC     Enter a number from the menu below to invoke or
                  cancel a condition handler.

                  Note that entering 3, 4, or 5 will end the lesson.

                  1 – signal CONDITION,1 handler (calls cdunit COND1)
                  2 – signal CONDITION,2 handler (calls cdunit COND2)
                  3 – execute CTRL_C handler (calls cdunit CTRLC)
                  4 – execute STOP handler (calls cdunit STOPL)
                  5 – execute ENDLESSON handler (calls cdunit ENDL)
                  6 – cancel a condition handler
.       INPUT
.       TEST      response
.       VALUE     "1"
.       .         SIGNAL    condition,1
.       VALUE     "2"
.       .         SIGNAL    condition,2
.       VALUE     "3"
.       .         LOOP      true
.       .         .         WRITC     Type a CTRL/C ...
.       .         .         PAUSE
.       .         ENDLOOP
.       VALUE     "4"
.       .         STOP
```

```
.          VALUE      "5"
.          OUTLOOP    true
.          VALUE      "6"
;
; Display a second menu of options.
;
.                     WRITC    Enter a number from the menu below
                               to cancel a condition handler.

                              1 – cancel CONDITION,1 handler
                              2 – cancel CONDITION,2 handler
                              3 – cancel CTRL_C handler
                              4 – cancel STOPLESSON handler
                              5 – cancel ENDLESSON handler

.          .          INPUT
.          .          TEST     response
.          .          VALUE    "1"
.          .          .        CANCEL   CONDITION,1
.          .          .        WRITC    The CONDITION,1 handler has been canceled.
.          .          VALUE    "2"
.          .          .        CANCEL   CONDITION,2
.          .          .        WRITC    The CONDITION,2 handler has been canceled.
.          .          VALUE    "3"
.          .          .        CANCEL   CTRL_C
.          .          .        WRITC    The CTRL_C handler has been canceled.
.          .          VALUE    "4"
.          .          .        CANCEL   STOPLESSON
.          .          .        WRITC    The STOPLESSON handler has been canceled.
.          .          VALUE    "5"
.          .          .        CANCEL   ENDLESSON
.          .          .        WRITC    The ENDLESSON handler has been canceled.
.          .          OTHER
.          .          .        WRITC    A handler has not been canceled.
.          .          ENDTEST
.          ENDTEST
ENDLOOP

CDUNIT     cond1
WRITC      You are now in the COND1 condition unit.
           The CONDITION,1 condition handler transferred
           control of the lesson to this condition unit.
           When you press RETURN, control will pass back to
           the menu of options display.

           ... Press RETURN to continue ...
PAUSE
```

```
CDUNIT   cond2
WRITC    You are now in the COND2 condition unit.
         The CONDITION,2 condition handler transferred
         control of the lesson to this condition unit.
         When you press RETURN, control will pass back to
         the menu of options display.

         ... Press RETURN to continue ...

PAUSE

CDUNIT   ctrlc
WRITC    You are now in the CTRLC condition unit.
         The CTRL_C condition handler passed control of the lesson
         to this condition unit when you pressed CTRL/C. DAL
         has a built-in CTRL/C handler that terminates the lesson when
         CTRL/C is pressed. That handler will execute when you
         press RETURN.

         ... Press RETURN to terminate the lesson ...
PAUSE

CDUNIT   stopl
WRITC    You are now in the STOPL condition unit.
         The STOPLESSON condition handler passed control of the lesson
         to this condition unit when the STOP instruction was
         encountered. When you press RETURN, the STOP instruction
         will stop the lesson.

         ... Press RETURN to stop the lesson ...
PAUSE

CDUNIT   endl
WRITC    You are now in the ENDL condition unit.
         The ENDLESSON condition handler passed control of the lesson
         to this condition unit when the ENDLESSON instruction was
         encountered. When you press RETURN, the ENDLESSON instruction
         will end the lesson.

         ... Press RETURN to end the lesson ...

ENDLESSON
```

12

The VAX DAL Color Management System

12

The VAX DAL Color Management System

Chapters 3 and 5 each explained some of the rudiments of the VAX DAL color management system. This chapter describes the color management system in detail, explaining the system's component parts and their interaction. At lesson startup, DAL initializes the color system with eight basic colors that you can use as foreground or background colors in your displays. Depending on the color capabilities of the terminals you use, you may be able to access and use additional colors in your lessons, with from 4 to 16 different colors appearing on the terminal screen at the same time.

Topics discussed in this chapter include:

- Color system components
  - Color specifications
  - Terminal color palette
  - DAL color map
  - DAL color table
  - Color management instructions
- The default color system
- Color system modification

You cannot modify the color capabilities of the terminals you use to deliver your lessons. VAX DAL color management instructions modify the DAL color system components only. They do not expand the color capabilities of terminals. The instructions can, however, make full use of the color capabilities of the terminal. Authors should refer to Chapter 7 in this guide, entitled Terminal Management, to look up the color capabilities of the terminals their lessons use.

THE COMPONENTS OF THE DAL COLOR MANAGEMENT SYSTEM

The DAL color management system consists of color specifications, three color structures, and five color management instructions. The three color structures are:

- The color palette of the terminal
- The DAL color map
- The DAL color table

DAL color management instructions send a color specification to the terminal whenever the author requests a color.

Color Specifications

A color specification is a code the terminal must use in order to generate a particular color. Terminals are equipped with a fixed number of color specifications. This number specifies the number of different colors the terminal is capable of generating. A terminal equipped with eight color specifications can generate only eight colors.

DIGITAL terminals use color specifications that are encoded by two different methods. The standard method is the Hue/Lightness/Saturation (HLS) method for specifying colors. The HLS method uses different values for hue, lightness, and saturation to specify a large range of colors. DIGITAL VT125, VT240, VT241 terminals, and DECmate III, Rainbow, and Professional terminal emulators use the HLS method of specifying colors.

DIGITAL terminals also use a color number method. With this method, each color specification is identified with an integer. DIGITAL GIGI (VK100) terminals use this method. The DECmate II terminal emulator uses both the HLS method and the color number method. (See Chapter 7: Terminal Models: Color Capabilities for more details.)

Terminal Color Palettes

Each of the terminals mentioned above stores color specifications in an internal color palette. A terminal color palette can be thought of as a storage cabinet containing cubbyholes or slots. Each slot contains a different color specification.

The number of slots in the color palette determines the number of color specifications the terminal can use. For example, the GIGI color palette has eight slots; consequently, GIGI terminals are equipped with eight different color specifications. Rainbow terminals have 4096 different color palette slots, hence, 4096 different color specifications.

For terminals that use the HLS method for specifying colors, the address of the color palette slot that contains an HLS specification is the HLS specification itself. For example, the slot that contains the specification 120 50 100 (the specification for the color red) has 120 50 100 as its address.

For terminals that use the color number method for specifying colors, the address of a slot containing a color specification is the color number identified with that specification. In the color number method, the color specification for red is identified with the number 2. Consequently, slot number 2 in the color palette contains the color specification for red.

For definitive information about the color palette of the terminal you use, you must refer to the Programmer's Reference Manual that comes with the terminal. The Programmer's Reference Manual provides a list of the color specifications your terminal is equipped with. You need information from this list to access and use the colors in your terminal color palette.

The DAL Color Map

Like a terminal color palette, the DAL color map stores color specifications. The color map is, in fact, the DAL counterpart to the terminal color palette. In the course of the lesson, color specifications are copied from the terminal color palette into the DAL color map.

Unlike the terminal color palette, where the number of storage slots determines the number of specifications the terminal has, the DAL color map is of a constant size, regardless of the terminal model DAL is executed on. The DAL color map has 64 color slots. These slots are addressed by number, from 0 to 63.

When the lesson uses a foreground color or background color, it refers to the DAL color map for color specifications, not to the terminal color palette. When a color is requested with an FCOLOR or BCOLOR instruction, DAL searches the DAL color map for the color specification of the requested color. When DAL finds the color specification, DAL sends it to the terminal color generation facility where the specification is used to generate the color given to subsequent text or graphics. Note that, because the color map contains only 64 slots, the color map can contain a maximum of 64 color specifications. Lessons, consequently, cannot use more than 64 different colors at one time.

Color specifications are loaded from the terminal color palette into the DAL color map by two different means. The first is automatic. The second is entirely under the author's control.

At lesson startup, DAL automatically loads the first eight slots in the color map (slots 0 through 7) with the color specifications of eight basic colors. This provides the author with enough colors for most simple lessons. The DAL-provided colors, their color specifications, and the numbers of the color map slots the specifications occupy, are listed below.

Table 12–1: DAL-Provided Color Specifications

Name	Color Specification (HLS)	Slot Number
DARK	0 0 0	0
BLUE	0 50 100	1
RED	120 50 100	2
MAGENTA	60 50 100	3
GREEN	240 50 100	4
CYAN	300 50 100	5
YELLOW	180 71 100	6
WHITE	0 100 100	7

As explained in earlier chapters, the FCOLOR and BCOLOR instructions bring new foreground and background colors into use. These instructions can specify the DAL-provided colors by color name or by color map slot number.

The reason DAL-provided colors can be invoked either by color name or by slot number is important for future discussion. DAL defines each of the DAL-provided color names as a constant equal to a slot number. Slot number 2, for example, is identified with the string constant "RED". Whenever an FCOLOR RED instruction is issued, DAL looks in slot number 2 in the color map for the color specification for red. Because the string RED equals 2 by definition, FCOLOR RED and FCOLOR 2 have the same effect.

The second way color specifications are loaded into the DAL color map is with the MAP instruction. Authors can use the MAP instruction to copy the specification of a color they need into a slot in the DAL color map from the terminal color palette. Once in the color map, the color can be requested by an FCOLOR or BCOLOR instruction that uses the color's slot number as its argument. (More on the MAP instruction follows.)

The DAL Color Table

The DAL color table is another color structure used by the DAL color management system. Unlike the color palette or map, the color table does not store color specifications. Instead, the color table acts as DAL's guide to the color map. The color table stores directional pointers that DAL uses to locate color specifications in the color map. The color table also limits the number of colors that DAL can use simultaneously.

Several of the terminals used to execute DAL lessons cannot support more than four different colors on the screen simultaneously. DAL acknowledges this restriction by limiting the number of slots in the DAL color table to the number of colors that the terminal can support on the screen at the same time. If DAL can locate (by referencing the color table) only four color specifications at a time, then there is no danger of using more colors in a single display than can be supported by the terminal.

DAL establishes the size of the color table at lesson startup. Authors can change the default size of the color table with the SET MAXCOLORS instruction, which is described later in this chapter. The size DAL assigns the color table depends on the terminal model in use. The table below displays the color table sizes that are used with different terminal models.

Table 12–2: Terminal Models: Color Table Size

Terminal Model	Color Table Size
VT125	4 slots
VT240/VT241	4 slots
DECmate III	4 slots
RAINBOW	4 slots
GIGI (VK100)	8 slots
PROFESSIONAL (with PRO/Communications V2.0)	8 slots
DECmate II	16 slots

Authors can use the SET MAXCOLORS instruction to reduce the size of the color table (from 16 slots down to 4, for example) in order to create a lesson that runs successfully on terminals with differing color capabilities. Authors cannot, however, successfully use SET MAXCOLORS to expand the color table to a size greater than the terminals in use can support.

DAL places a new pointer in the DAL color table each time a new foreground or background color is used. The pointer directs DAL to a color specification in the color map.

The first FCOLOR instruction executed in a lesson sets a pointer in slot number (MAXCOLORS–1) in the color table. The second FCOLOR instruction sets a pointer in slot number (MAXCOLORS–2), and the third in slot number (MAXCOLORS–3). Color table slot number 0 always contains the pointer to the specification for the current background color. Whenever a new background color is specified, the pointer to the current background color specification is erased from slot 0, and a pointer to the specification for the new background color replaces it.

If you reuse a color that you have already used on the screen, DAL looks up the pointer for that particular color in the color table. If DAL finds the pointer, DAL uses it to find the required color specification.

After a third new foreground color is used at a terminal that supports only four colors on the screen, the color table is full (the fourth color is the background color). Each slot in the table contains a pointer, even if only one foreground color was used at a time. After the color table is full, subsequent FCOLOR instructions are ignored because DAL has no place to set pointers for any additional foreground colors. Authors must use either the FCOLOR instruction with the table_slot_number argument, or the CCOLOR instruction (see below) to bring new foreground colors into use when the color table is full.

THE VAX DAL COLOR MANAGEMENT INSTRUCTIONS

The five VAX DAL color management instructions are the FCOLOR, CCOLOR, BCOLOR, MAP, and SET MAXCOLORS instructions. These instructions manage the DAL color system, bringing color specifications into use and setting limits on the color system. This section explains the function of each of the instructions in detail.

The FCOLOR Instruction

The FCOLOR instruction specifies the color of subsequent text or graphics. FCOLOR uses two different syntaxes:

```
FCOLOR color_constant_name{, table_slot_number}

FCOLOR map_slot_number{, table_slot_number}
```

Where:

color_constant_name is one of the DAL-provided color constants listed in Table 12–1. Because these constants are equated with slot numbers in the DAL color map, the two FCOLOR syntaxes are more alike than they appear. Note that authors can also define string constants equal to slot numbers.

map_slot_number specifies a slot in the DAL color map that holds a color specification. FCOLOR 7, for example, makes the color that is defined by the specification in slot number 7 the new foreground color.

{*table_slot_number*} specifies a slot in the DAL color table. DAL sets the pointer to the requested color specification in the color table slot specified by the argument. And, if a pointer was previously in that slot, DAL replaces it with the new pointer.

Using the FCOLOR instruction with the table_slot_number argument is one way to request additional colors when the color table is full. The instruction removes the pointer from the color table slot specified, and sets a new pointer in the slot for the new color specification. Be aware that the current color whose pointer is replaced becomes unavailable: if any graphics or text that use the current color are still on the screen, they are changed to the new foreground color.

The FCOLOR instruction functions in three different ways, depending on the state of the DAL color table and the syntax of the instruction:

- If color table slots are available, an FCOLOR instruction sets a pointer to the required color specification in the first available slot, or in the slot specified by the table_slot_number argument. Subsequent text and graphics use the new foreground color.
- If the color table is full and an FCOLOR instruction is used without a table_slot_number argument, the FCOLOR instruction is ignored. Subsequent text and graphics use the foreground color specified by the last successful FCOLOR instruction.
- If the color table is full and the FCOLOR instruction is used with a table_slot_number argument, the current pointer in the specified color table slot is replaced by a new pointer. Subsequent text and graphics use the new foreground color.

The CCOLOR instruction

The CCOLOR instruction clears any or all of the pointers from the DAL color table. This is another way to make new colors available after the color table reaches capacity. The CCOLOR instruction has three syntaxes:

CCOLOR color_constant_name

CCOLOR table_slot_number

CCOLOR ALL

Where:

color_constant_name is the color name that identifies the color specification you no longer need. CCOLOR RED, for example, removes the pointer to the color specification for red from the color table. Because DAL equates the string "RED" with slot number 2 in the color map, a CCOLOR RED instruction eliminates the pointer in the color table that points to color map slot number 2. This assumes that slot number 2 still contains the color specification for red. CCOLOR RED makes unavailable for displays the color defined by the specification in slot number 2 in the color map (unless another pointer to slot number 2 is created). A CCOLOR instruction that specifies a color_constant_name that has not been defined (either by DAL at lesson startup or by the author) has no effect.

table_slot_number specifies a slot in the DAL color table that is to be emptied. CCOLOR 1 removes the pointer from color table slot number 1 and makes the color pointed to from that slot unavailable.

ALL is a keyword that clears the entire color table. After a CCOLOR ALL instruction, the background color defaults to DARK and the foreground color is undefined. A foreground color must be specified before any text or graphics can be written to the screen.

The BCOLOR Instruction

The BCOLOR instruction specifies a color for the background of a display. The BCOLOR instruction automatically clears the current pointer from color table slot number 0 and creates a new pointer to the color specification of the new background color. The BCOLOR instruction has two syntaxes:

BCOLOR color_constant_name

BCOLOR map_slot_number

The parameters for the BCOLOR instruction match those for the FCOLOR instruction. Note, however, that the BCOLOR instruction never requires the table_slot_number argument that can be used with FCOLOR instructions. BCOLOR automatically clears the current pointer from the color table slot it updates.

The pointer to the color specification for the current background color always occupies color table slot number 0. DAL ignores an FCOLOR instruction that tries to place a pointer in slot number 0.

The MAP Instruction

The MAP instruction makes all of the color specifications in a terminal color palette available to DAL. The MAP instruction copies color specifications from the terminal color palette into slots in the DAL color map. Once in the color map, the color specifications can be requested with FCOLOR map_slot_number and BCOLOR map_slot_number instructions.

The Programmer's Reference Guide for your terminal model lists the specifications contained in your terminal color palette. Before you can use the MAP instruction, you must find out what specifications your terminal has.

If your terminal uses the HLS method for storing color specifications (all but GIGI terminals do), use the following MAP instruction syntax:

```
MAP map_slot_number, gray_value, H_value, L_value, S_value
```

Where:

map_slot_number is the slot in the DAL color map in which the color specification is to be loaded. The color map consists of 64 slots, numbered from 0 to 63. This means that a lesson can use a maximum of 64 different colors. Remember that the eight basic DAL-provided colors are stored in slots 0 through 7. If you load a color specification into one of those slots, the color specification for the basic color becomes unavailable. An FCOLOR map_slot_number or BCOLOR map_slot_number instruction can be used to request a color whose specification is loaded into a slot in the color map.

gray_value establishes the shade of gray that replaces the color if the lesson is run on a black-and-white terminal. This argument is an integer from 0 to 100. A gray level of 0 means the color is replaced by black on a black-and-white terminal. Gray level 100 replaces the color with white, 75 with light gray, 50 with medium gray, and 25 with dark gray.

H_value, L_value, and S_value define the color specification that the MAP instruction moves from the color palette to DAL's color map. H_value corresponds to the hue value of the specification. L_value corresponds to the lightness value of the specification. S_value corresponds to the saturation value of the specification. You can find the HLS values for the colors you need listed in your terminal's Programmer's Reference Guide.

The following MAP instruction moves the color specification for firebrick red (available in VT240s and other terminals), and a gray level value of 75, into slot 45 in the DAL color map:

```
MAP 45,75,120,35,60
```

The color palette of a GIGI terminal contains only the eight colors that DAL uses to initialize the DAL color map. The MAP instruction therefore cannot move any additional colors into the DAL color map in a lesson executed at a GIGI terminal.

The MAP instruction has a second syntax, however, that can be used to rearrange the eight colors in a GIGI-DAL color map. This second syntax can also be used with other terminal models to move color specifications identified with color constant names to new locations in the color map. The second syntax for the MAP instruction is:

```
MAP map_slot_number, color_constant_name
```

For example, a MAP 19,RED instruction moves the specification identified with the color_constant_name RED to slot number 19 in the color map. Use this second MAP syntax when you need to place the color specifications in the color map into a particular sequence.

The SET MAXCOLORS Instruction

SET MAXCOLORS changes the default number of slots available in the DAL color table. DAL uses a SET MAXCOLORS instruction at lesson startup to match the color table size to the color capabilities of the terminal. You may wish to change the size of the color table if you develop lessons on 8- or 16-color terminals that must run on 4-color terminals, or the other way around. The syntax of the SET MAXCOLORS instruction is:

```
SET (TAB)/(SP) MAXCOLORS, value
```

Where:

value is the number of slots the color table is to have.

A COLOR MANAGEMENT EXAMPLE

The following source code example uses VAX DAL color instructions to manage the terminal color palette, DAL color map, and DAL color table. At appropriate times in the source code, a schematic diagram of the state of the color map and the color table is shown. Figure 12–1 shows the color map and color table at lesson startup.

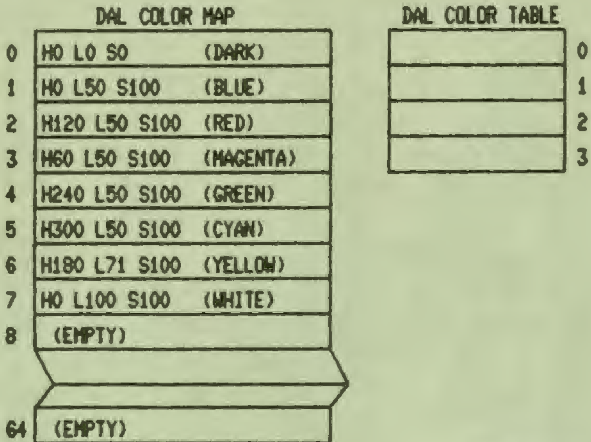


Figure 12–1
Default Color System: Lesson Startup

```
LESSON  DEMO
!
! This lesson adds four colors to the eight already provided by
! VAX DAL. First, the names of the new colors are defined as
! constants equal to slot numbers in the DAL color map.
!
DEFINE  coral = 8
DEFINE  thistle = 9
DEFINE  khaki = 10
DEFINE  orchid = 11
DEFINE  wheat = 12
!
! Next, gray levels & HLS specifications for the new colors are
! loaded into the DAL color map. Note that because the color names
! were defined as constants equal to numbers, we can use the color
! names in place of slot numbers.
!
MAP     coral,25,150,50,10
MAP     thistle,25,60,80,25
MAP     khaki,50,180,50,25
MAP     orchid,50,60,65,80
MAP     wheat,75,180,80,25
!
```

```
! Now change the color specification for the DAL-provided
! color GREEN to lime green.
!
MAP      green,50,240,50,60
!
```

Figure 12–2 shows the color map after the new color specifications are loaded into it.

DAL COLOR MAP

0	H0 L0 S0 (DARK)
1	H0 L50 S100 (BLUE)
2	H120 L50 S100 (RED)
3	H60 L50 S100 (MAGENTA)
4	H240 L50 S60 (LIME)
5	H300 L50 S100 (CYAN)
6	H180 L71 S100 (YELLOW)
7	H0 L100 S100 (WHITE)
8	H150 L50 S10 (CORAL)
9	H60 L80 S25 (THISTLE)
10	H180 L50 S25 (KHAKI)
11	H60 L65 S80 (ORCHID)
12	H180 L80 S25 (WHEAT)
13	(EMPTY)
64	(EMPTY)

DAL COLOR TABLE

	0
	1
	2
	3

MR-S-3943-85

Figure 12–2
Modified Color Map

Color Management Example (cont.): Lesson DEMO

```
! This lesson is written for a four-color terminal.
! SET MAXCOLORS 4 allows for one background color and three
! foreground colors.
!
SET       MAXCOLORS, 4
!
! Use WHEAT for the background color.
!
BCOLOR    wheat
!
! Point to the color specification for ORCHID from slot number 1
! in the color table.
!
FCOLOR    orchid,1
SIZE      2
AT        10
WRITE     This foreground color is ORCHID.
!
! Point to the color specification for THISTLE from slot number 2
! in the color table.
!
FCOLOR    thistle,2
AT        610
WRITE     This foreground color is THISTLE.
!
! Point to the color specification for CORAL from slot number 3
! in the color table.
!
FCOLOR    coral,3
AT        1210
WRITE     This foreground color is CORAL (press RETURN).
PAUSE
```

Figure 12–3 displays the first configuration of pointers in the DAL color table.

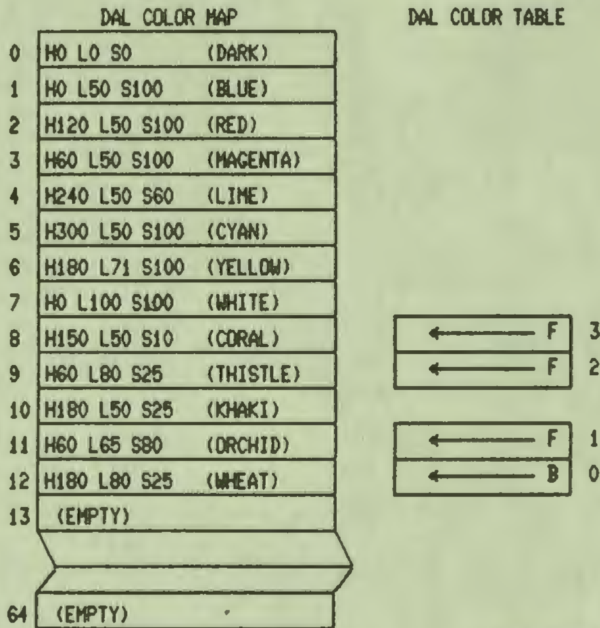


Figure 12–3
Initial Color Table Configuration

Color Management Example (cont.): Lesson DEMO

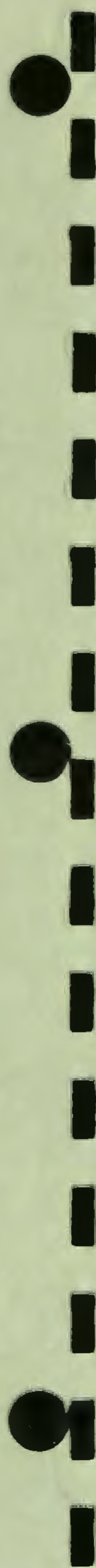
```
ERASE
!
! Now the color table is full (as MAXCOLORS = 4), so we'll clear it.
!
CCOLOR ALL
!
! Point to the color specification for WHEAT from the first
! available slot in the color table: slot number 3 (MAXCOLORS-1).
!
FCOLOR  wheat
AT      10
WRITE   This foreground color is WHEAT. Notice that
        the background is now dark--CCOLOR ALL
        cleared the entire color table and returned
        the background color to its default: dark.
!
! Point to the color specification for KHAKI from the next
! available slot in the color table: slot number 2 (MAXCOLORS-2).
!
FCOLOR  khaki
AT      610
WRITE   This foreground color is KHAKI.
!
! Point to the color specification for GREEN from the last
! available slot in the color table: slot number 1 (MAXCOLORS-3).
!
FCOLOR  green
AT      1210
WRITE   This foreground color is in GREEN's slot in
        the color map. It is actually lime green
        because the color specification in that slot
        was changed to lime green earlier.
PAUSE
ENDLESSON
```

Figure 12–4 displays the last configuration of pointers in the DAL color table.

	DAL COLOR MAP		DAL COLOR TABLE	
0	H0 L0 S0	(DARK)	←—— B	0
1	H0 L50 S100	(BLUE)		
2	H120 L50 S100	(RED)		
3	H60 L50 S100	(MAGENTA)		
4	H240 L50 S60	(LIME)	←—— F	1
5	H300 L50 S100	(CYAN)		
6	H180 L71 S100	(YELLOW)		
7	H0 L100 S100	(WHITE)		
8	H150 L50 S10	(CORAL)		
9	H60 L80 S25	(THISTLE)		
10	H180 L50 S25	(KHAKI)	←—— F	2
11	H60 L65 S80	(ORCHID)		
12	H180 L80 S25	(WHEAT)	←—— F	3
13	(EMPTY)			
64	(EMPTY)			

MR-S-3945-85

Figure 12–4
Final Color Table Configuration



13

File Input/Output in VAX DAL

13

File Input/Output in VAX DAL

Authors can use VAX DAL file input/output (I/O) instructions to access up to 16 different external files in their lessons. File I/O instructions can be used to:

- Access or create external files
- Deposit or retrieve data in external files
- Locate, delete, or update data in external files

Files can be used as a lesson database or as a repository for student comments or questions.

This chapter explains the use of external files in a DAL lesson. Topics discussed in this chapter include:

- Principal data elements, file structures, and record access modes used with external files
- VAX DAL instructions that authors use to access and modify data stored in files
  - OPEN
  - GET
  - PUT
  - FIND
  - UPDATE
  - DELETE
  - CLOSE

- VAX DAL file I/O operations with sequential files
- VAX DAL file I/O operations with random files
- VAX DAL file I/O operations with indexed files

Authors already familiar with basic file I/O concepts may want to skip directly to the sections dealing with file I/O in VAX DAL.

DATA ELEMENTS

In descending order of size, the basic units of data in any file are records, data fields, and bytes. Files store data in these units. The principal unit of data in a file is a record. Records are usually made up of data fields, and data fields, in turn, are made up of bytes.

Records

A record is a collection of several pieces of data that are treated as a whole. Examples are a person's home address or a personnel record. In the address example, a person's full name is considered one piece of data in the data collection. Street address, city, state, and zip code are other pieces of data. Taken separately, each piece of data is of marginal usefulness. Together in a single record, however, the pieces form one useful, manageable bundle of information.

Data Fields

Data fields isolate the different pieces of data contained in a single record. To carry on with the above address example, an individual's last name is one data field in a record that contains that individual's address. Another data field contains the individual's street address, yet another, the zip code. Data fields, then, are different strings of characters that are parts of a single record. The characters in a data field are stored as bytes of data.

Bytes

A byte is a group of eight binary digits used to represent a single character. The name Sam, for example, requires a field in a record at least three bytes in length (one byte for each character in the name).

FILE STRUCTURES

Records, which are made up of fields and bytes, are stored in files. A file is a structured collection of records kept on a storage medium, such as a disk or a magnetic tape.

The structure of a file determines how the records in the file are stored, and how records can be retrieved from or added to the file. What follows is a discussion of three frequently encountered file structures.

Sequential File Structure

In a sequential file, records are stored in the order in which they are added to the file. The address of a record (the location at which the record can be found in the file) is determined by its place in the sequence of records. The third record added to the file, for example, is record number three. The fourth record added becomes record number four. The location of a record in the sequence of records gives the record its record number and address. Thus, records cannot be inserted between records already stored in a sequential file because inserting a record in the middle of the file would disrupt the existing record sequence.

Sequential files are useful for storing records that have varying lengths. For example, the first record in the file can be 64 bytes long. The second record can be 128 bytes long. Records in sequential files are not required to have a uniform size.

Access to Records in Sequential Files

The location of each record in a sequential file is specified by its place in the sequence of records. To find a particular record, the computer must be given the number of the record. The computer then reads through and counts all of the records preceding the desired record in the file. For example, to find the fifth record in a sequential file, the computer starts at the beginning of the file and reads to the end of the fourth record. After the fourth record, the computer finds the fifth record.

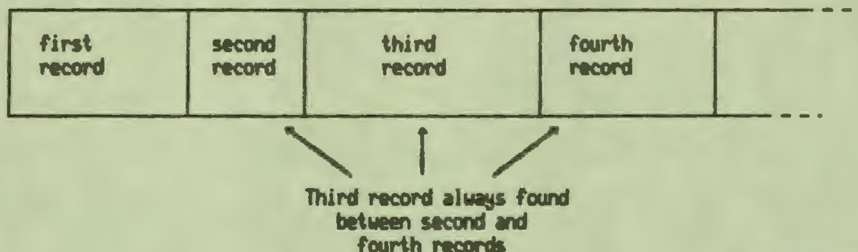


Figure 13–1
Sequential File Organization

Random File Structure

In a random file, each record occupies a numbered location called a relative cell. Each time a record is inserted in a relative cell, the record is assigned the number of the cell as its address, or *relative record number*. A relative record number specifies the location of the record relative to the beginning of the file.

In random files, programmers can leave empty spaces between records. For example, records can be placed in the ninth and eleventh cells of a random file while the tenth cell is left empty. The programmer can later place a record in the tenth cell.

Records are limited to a specific size in random files. They cannot be longer than the relative cells they are to occupy. If a record is too long or too short for the relative cell, the record is truncated or blank padded to make it the correct size.

Access to Records in Random Files

Because each cell in a random file is of a fixed length, the location of a record in the file can be calculated from its relative record number. The computer multiplies the relative record number by the length of each cell to determine where the desired record is in relation to the beginning of the file. Control is then transferred directly to the desired record. This process is faster than using the sequential-access method in which each record preceding the desired record in the file must be read before the desired record can be found.

Records in random files can be accessed sequentially, just as records are accessed in sequential files. Authors can use sequential access methods to read through each record preceding the desired record in a random file.

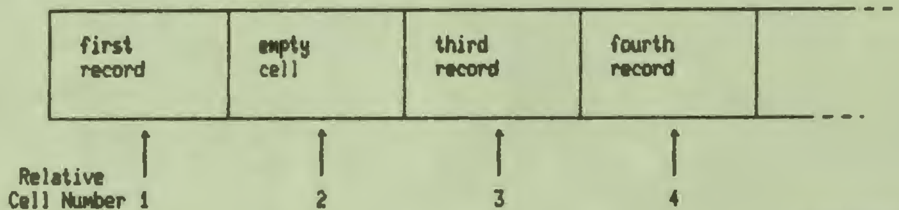


Figure 13–2
Random File Organization

Indexed File Structure

Records are indexed when they are added to an indexed file. Programmers designate one or more data fields in each record as the *keys* to that record. The computer then creates indexes from the keys to the records.

The index of an indexed file is much like the index of a book. Just as a reader refers to a book index to locate needed information, a computer refers to a file index to locate a needed record.

As each record is added to an indexed file, the contents of one or more data fields are copied from the record into the file index. These data fields become the keys to the records. Each key in the index is listed with the location of the record of which the key is a part. Given a key, the computer can locate an entire record.

Programmers must specify one data field in the record format as the *primary key*. The principal index to the file is constructed of the primary keys from the records in the file. In the address record example used earlier, an individual's last name is listed as one data field in the individual's address record. The last name data field is frequently used as the primary key to the record. Given a person's last name, the computer can locate the person's entire address record.

Programmers can designate other data fields in the records as the records' secondary keys. The computer uses the secondary keys to create secondary or subindexes to the records in the file.

Access by Key Value to Records in Indexed Files

Because the location of each record in an indexed file is documented in the file indexes, all that is necessary to locate a particular record in an indexed file is the entry for that record in the primary index, and optionally, any entries the record may have in secondary indexes.

The index to an indexed file is made up of the keys to the records in the file. A key is the actual contents of one data field from a record in the file. If you know the key to a record that is listed in the file index, you can find the record in the file.

Records in indexed files can be accessed sequentially. Authors can access the records in a file one after another, in the order in which they are listed in the file index.

For more information about the structure and elements of files, consult the *Guide to VAX/VMS File Applications* in the VAX/VMS documentation set.

THE VAX DAL FILE I/O INSTRUCTIONS

Several steps must be taken to access, read, and modify data in a file external to a DAL lesson. Each step is slightly different depending on the structure of the external file you wish to use. The seven instructions provided by DAL that allow the use of external files are:

- OPEN
- GET
- PUT
- FIND
- UPDATE
- DELETE
- CLOSE

First, the function of each instruction is explained. Next, the syntax of each instruction is explained in the context of its use with a particular file structure.

The OPEN Instruction

The OPEN instruction establishes contact with the file to be used. It must precede all other file-processing instructions. The OPEN instruction does the following:

- Establishes the channel number used to access and identify the file in subsequent GET, PUT, UPDATE, DELETE, and CLOSE instructions.
- Finds the requested file, opens it, and verifies its identity as the requested file.

- Establishes an access mode for the file. Possible access modes are:
  - READ — the file can only be read from
  - WRITE — the file can only be written to
  - UPDATE — the file can be both read from and written to

If the requested file cannot be found, the OPEN instruction creates a file with the name and attributes that the OPEN instruction specifies.

The GET Instruction

The GET instruction reads a record from a file into a string variable or record structure. The lesson can then use the variable or structure that contains the record. The GET instruction can be used in two different ways. Depending on the syntax used, GET either reads the current record from the file, or locates and reads a specific record from the file.

The PUT Instruction

The PUT instruction writes a record from a string variable or record structure into an external file. Where the record is written in the file depends on file structure and on the syntax of the PUT instruction.

The FIND Instruction

The FIND instruction locates a record in a file. While FIND does not read a record it locates into a variable or record structure, it does make the record the current record in the file. As the current record, the record can be read by the GET instruction, erased by the DELETE instruction, or modified by the UPDATE instruction. The FIND instruction cannot be used with sequential files.

The UPDATE Instruction

The UPDATE instruction replaces the contents of the current record in the file with the contents of a record structure declared at lesson or module level. UPDATE can only be used with indexed files.

The DELETE Instruction

The DELETE instruction, as its name implies, deletes the current record in a file. DELETE also can only be used with indexed files.

The CLOSE Instruction

The CLOSE instruction closes a file so that its contents are no longer available to a lesson. Depending on its syntax, the CLOSE instruction either closes or deletes the specified file. The syntax of the CLOSE instruction is the same regardless of file structure. The syntax of the CLOSE instruction is:

CLOSE (TAB)/(SP) channel_no{, DELETE}

Where:

channel_no is the identifying number assigned to the file by the OPEN instruction that accessed the file.

{DELETE} is an optional argument that deletes as well as closes the specified file.

FILE I/O OPERATIONS WITH SEQUENTIAL FILES

An OPEN instruction must precede all other file I/O operations involving a sequential file. The syntax of the OPEN instruction used to access or create a sequential file is:

OPEN (TAB)/(SP) file_name, channel_no, mode, SEQUENTIAL

Where:

file_name specifies the name of the file.

channel_no is a number from 0 to 15 used to identify the file in subsequent file I/O operations.

mode is a keyword that specifies an access mode for the file (the access mode keywords are: READ, WRITE, UPDATE).

SEQUENTIAL is the keyword that specifies the file structure of the file being opened.

Reading Records from a Sequential File

The syntax of the GET instruction used to read records from a sequential file is:

```
GET(TAB)(SP) channel_no, variable_name
```

Where:

channel_no is the identifying number given to the file by the OPEN instruction that accessed it.

variable_name specifies the name of a variable. This variable must be of the same data type as the data in the records.

Two things affect reading records from sequential files. First, control is positioned at the beginning of the file when a sequential file is opened in READ-access mode. This means that, when the file is first accessed, the first record in the file is the current record.

Second, the GET instruction can only read the current record from a sequential file. GET cannot position the file at a particular record and then read that record. To read a record other than the first record in a sequential file, you must first use system functions to position the file at the record you need.

Positioning a file involves moving control forward or backward through the file until the desired record is reached. The record at which control is positioned becomes the new *current record* in the file. Two different system functions can be used to position control at a record in a sequential file for a GET operation: the FIND system function and the REWIND system function. You cannot use the FIND instruction to position a sequential file.

The FIND system function attempts to position control at a specified record in a file. The syntax of the FIND function is:

```
FIND(channel_no,record_no)
```

Where:

channel_no is the channel number of the file containing the desired record.

record_no is the record number of the desired record in the file.

If the FIND function successfully positions the file at the specified record, the function returns the value 1. If FIND is not successful, it returns 0 and the IORESULT variable is set with a value that indicates why the attempt failed. FIND automatically invokes the REWIND system function (see below) if the record number of the desired record is lower than the record number of the current record.

A GET instruction executed after a successful FIND system function reads the current record from a sequential file. After a successful GET operation, the next record in the file becomes the current record. Because of this, successive GET instructions can read all of the records in a sequential file, in the order they fall in the file.

The REWIND system function positions control at the first record in the file. The syntax of the REWIND function is:

```
REWIND(channel_no)
```

Where:

> *channel_no* is the channel number of the file that contains the desired record.

If a GET instruction is executed after a REWIND, the first record in the file is read into the variable named in the syntax of the GET instruction.

The DAL code below shows how the GET instruction and system functions can be used to read a record from a sequential file:

```
DEFINE    status : integer
DEFINE    ans14 : string
OPEN      "answers.dat",3,read,SEQUENTIAL
ASSIGN    status := FIND(3,14)
IF        status = 1              $$ successful FIND operation
.         GET       3,ans14
.         WRITC     The correct answer is <<s,ans14>>
ELSE
.         WRITC     The correct answer could not be located
ENDIF
```

Writing Records to a Sequential File

The syntax of the PUT instruction used to write records into a sequential file is:

```
PUT (TAB)/(SP) channel_no, variable_name
```

Where:

> *channel_no* is the identifying number given to the file by the OPEN instruction that accessed it.
>
> *variable_name* specifies the name of the string variable containing the data to be written. Data can only be written to a sequential file from a variable with the string data type.

Records can be written to a sequential file only at the end of file (EOF) position. If you attempt to add a record to the file anywhere but at EOF, nothing is written; the RMSSTATUS variable is set to 98812, and IORESULT is set to 10.

Because a sequential file is positioned at EOF when it is accessed in WRITE mode, you can successfully write records to a newly opened sequential file. If the file is positioned anywhere other than at EOF by earlier operations that use the FIND or REWIND functions, use the EOF system function to reposition the file. The syntax of the EOF system function is:

```
EOF(channel_no)
```

Where:

> *channel_no* is the identifying number of the file in which you would like to write the record.

The DAL code below displays how the PUT instruction can be used to write records into a sequential file.

```
DEFINE   comment:string
DEFINE   status:integer
OPEN     "comments.dat",4,write,SEQUENTIAL
AT       510
WRITE    Do you have any comments about the material we
         have covered so far? If so, go ahead and enter
         them at the prompt. If not, press RETURN.
INPUT    515
ASSIGN   comment := RESPONSE
IF       comment <> ""
.        ASSIGN   status := EOF(4)     $$ to ensure that
.        PUT      4,comment            $$ the file is at EOF
ENDIF
```

FILE I/O OPERATIONS WITH RANDOM FILES

An OPEN instruction must precede all other file I/O operations involving a random file. The syntax of the OPEN instruction used to access or create a random file is:

```
OPEN (TAB)/(SP) file_name, channel_no, mode, RANDOM, record_size
```

Where:

file_name specifies the name of the file.

channel_no is a number from 0 to 15 used to identify the file in subsequent file I/O operations.

mode is a keyword that specifies an access mode for the file (the access mode keywords are: READ, WRITE, UPDATE).

RANDOM is the keyword that specifies the file structure of the file being opened.

record_size specifies the maximum number of bytes (characters) that each record can contain. This number also specifies the size of the relative cells in which the records are stored.

If an OPEN instruction specifies a random file that does not already exist, a file is created with the characteristics (file name, access mode, and record size) specified by the instruction. Records can then be written to the newly created file.

Reading Records from a Random File

The syntax of the GET instruction used to read records from a random file is:

GET (TAB)/(SP) channel_no, variable_name{, record_number}

Where:

channel_no is the identifying number given to the file by the OPEN instruction that accessed it.

variable_name specifies the name of a string variable.

{*record_number*} is the number of the record to be read.

A GET instruction can both locate and read a record from a random file. By multiplying the record number by the maximum record size specified in the OPEN instruction that accessed the file, GET calculates the position of the record relative to the beginning of the file. The GET instruction then writes the contents of the record into the variable the GET instruction specifies.

The REWIND and FIND system functions and the FIND instruction can also be used to position a random file at a record. A GET instruction without a record number argument reads the current record from a random file.

After GET reads a record from a random file, the next sequential record in the file becomes the current record. The next record in the file is positioned as the new current record even if there are empty cells between the last record read and the new current record.

The FIND instruction can be used to locate a record in a random file. The FIND instruction can only locate a record and make it the current record in the file. FIND does not modify, read from, or write to the record in any way.

The syntax of the FIND instruction used with a random file is:

```
FIND (TAB)/(SP) channel_no, record_no
```

Where the parameters are the same as the corresponding parameters used with the GET instruction for a random file.

The DAL code below displays one application of the instructions used to locate and read a record from a random file.

```
DEFINE  ans14 : string
OPEN    "answers.dat",5,read,RANDOM,60
GET     5,ans14,14
IF      IORESULT = 1
.       WRITC   The correct answer is <<s,ans14>>
ELSE
.       WRITC   The correct answer could not be located
ENDIF
```

Writing Records to a Random File

The syntax of the PUT instruction used to write records to a random file is:

```
PUT (TAB)/(SP) channel_no, variable_name{, record_no}
```

Where:

> *channel_no* is the identifying number given to the file by the OPEN instruction that accessed it.
>
> *variable_name* specifies the name of the string variable or record structure containing the data to be written. Data can be written only from variables that have the string data type. Component variables in record structures used with random files can be of any data type except table or record. See the next section in this chapter for more information about record structures.

{record_no} is the number of the relative cell in the file into which the record is written. If a relative cell is not specified, the record is written into the cell whose relative record number is one higher than the number of the last cell written to.

The contents of the variable named by the PUT instruction are written into the specified relative cell in the file. PUT instructions can insert records into empty cells between existing records in a random file. For example, if records are written to cell number 5 and cell number 7 in the file, a record can later be inserted in cell number 6.

Records cannot be written to cells that already contain records.

Records of varying lengths can be written to fixed-length cells. If the record is too long, it is truncated. If the record is too short, it is blank padded. Authors can use the LEN function to determine the number of characters in a string variable, or the total number of characters in the component variables of a record structure. The syntax of the LEN system function is:

```
LEN(var_name)
```

Where:

var_name is the name of a string variable or record structure.

Random files can be positioned at EOF for a write by the EOF system function.

The following DAL code displays how the PUT instruction can be used to write records to a random file.

```
UNIT put_average
;
;        Record the student's average up to this point in
;        the lesson.
;
DEFINE   average : string
DEFINE   i : integer
OPEN     "averages.dat",4,write,RANDOM,5
ASSIGN   average := string(NOK/QUERIES)*100
ASSIGN   i := i+1
PUT      4,average,i
```

FILE I/O OPERATIONS WITH INDEXED FILES

An OPEN instruction must precede all other file I/O operations involving an indexed file. The syntax of the OPEN instruction used to access or create an indexed file is:

```
OPEN (TAB)/(SP) f_name,ch_no,mode,INDEXED,rec_name,shr_type,prim_key_name
                 {sec_key_0_name{, sec_key_n_name...}}
```

Where:

f_name specifies the name of an indexed file.

ch_no is a number from 0 to 15 used to identify the file in subsequent file I/O operations.

mode is a keyword that specifies an access mode for the file (the access mode keywords are: READ, WRITE, UPDATE).

INDEXED is the keyword that specifies the file structure of the file being opened.

rec_name is the name of a record structure defined at lesson or module level.

shr_type is a keyword that specifies the type of file sharing allowed (possible keywords are READ, WRITE, UPDATE, or NONE).

prim_key_name specifies the name of a component variable in the rec_name record structure. This component variable corresponds to the primary key data field in the records.

{*sec_key_0_name*} specifies the name of a component variable in the rec_name record structure. This component variable corresponds to the first secondary key data field in the records.

{*sec_key_n_name*} specifies the name of a component variable in the rec_name record structure. This component variable corresponds to the nth secondary key data field in the records.

Enter the OPEN instruction and all its parameters on one line in the source code.

If an OPEN instruction specifies an indexed file that does not already exist, an indexed file is created with the characteristics specified by the instruction. Records can then be written to the newly created file.

Record structures are used in many file I/O operations involving indexed files. All data written to, and all data read from indexed files is passed by means of record structures.

A record structure is a collection of related variables, just as a record is a collection of related fields of data. Each component variable in a record structure corresponds to one or more fields of data in the records of a file. Component variables must be of the same data type as the data field to which they correspond. Component variables cannot have the TABLE or RECORD data type; the nesting of one record structure within another is not supported. Define the component variables in an order that corresponds to the order of the data fields in the record.

String variables defined in record structures must specify a fixed length for strings. Because each field in a record is of a fixed length, the string variable that corresponds to a field in the record must be of a fixed length. The syntax of the DEFINE instruction used to define a fixed-length string is:

```
DEFINE (TAB)/(SP) variable_name : STRING, length
```

Where:

> *length* specifies the maximum number of bytes (characters) allowed in the string.

A variable-length string can be assigned to a fixed-length string variable. The string is either blank padded or truncated to fit the fixed length. Fixed-length strings can be declared only in record structure definitions.

The name of a variable defined as part of a record structure must be unique. The name of the record structure itself must also be unique.

The DAL code below displays an example of a record structure definition. Each component variable corresponds to one field of data in the records of an indexed file. The component variables must be defined in the same order as the fields to which they correspond in the record.

```
DEFINE   student_data : RECORD
.        DEFINE   last_name : STRING,25
.        DEFINE   first_name : STRING,15
.        DEFINE   class_rank : INTEGER
.        DEFINE   GPA : REAL
.        DEFINE   test_score[5] : INTEGER
.        DEFINE   meeting : BOOLEAN
ENDRECORD
```

The last instruction in a record definition must be the ENDRECORD instruction. ENDRECORD signifies the end of the record definition for the compiler.

Reading Records from an Indexed File

The of the GET instruction used to read records from an indexed file is:

GET (TAB)/(SP) channel_no, key_number, { EQ | GT | GE }, key_value

Where:

channel_no is the identifying number given to the indexed file by the OPEN instruction that accessed it.

key_number is a number that specifies which index (key) to use to locate the record (primary key = 0, first secondary key = 1, nth secondary key = n).

{ *EQ* | *GT* | *GE* } specifies which of three types of key value match you require.

key_value is the data field in a record that acts as the key to that record.

Depending on the syntax you use, the GET instruction either locates and reads a specified record from an indexed file, or reads the current record from an indexed file. In both cases, GET copies the record into the record structure specified by the OPEN instruction that accessed the file.

A GET instruction that uses all of the parameters listed above locates and reads a record from an indexed file. The key number argument indicates whether the key value you specify is in the primary or secondary indexes. The key value argument specifies the actual index entry that identifies the record you need. Given these two arguments, the GET instruction can locate the record.

You can specify that you want a record with a key value that is equal to (EQ), greater than (GT), or greater than or equal to (GE) a value you specify. This applies to both numeric and alphabetic key values. The letter A has a lower value than the letter Z.

Once the required record is located, the GET instruction assigns the first field of data in the record to the first component variable in the record structure named in the OPEN instruction that accessed the file. Next, it assigns the second field of data to the second variable of the record structure and so on, until the entire record is copied into the record structure.

Records in indexed files can be accessed and read sequentially. The order assigned the records in an indexed file depends on the ordering of the records' keys in the indexes to the file. Records are listed in file indexes in the ascending order of their key values. If the key to each record is numeric, the record with the lowest key value is the first record listed in the index. If the key to each record is alphabetic, the records are listed in alphabetic order in the index. A GET instruction with just a channel number as its argument reads the current record from the file. After a GET operation in an indexed file, the record listed next in the index becomes the current record.

The DAL code below displays how records can be read from indexed files.

```
LESSON    show_stat
DEFINE    student_data : RECORD
.         DEFINE    last_name : STRING,25
.         DEFINE    password : STRING,5
.         DEFINE    class_rank : INTEGER
.         DEFINE    GPA : REAL
.         DEFINE    test_score[4] : INTEGER
.         DEFINE    meeting : BOOLEAN
ENDRECORD
AT        510
WRITE     Would you like to see your current
          status in this class? [Yes/No]
QUERY     *
RIGHT     Yes | Y
.         DO        status
RIGHT     No | N
.         BRANCH    $END
WRONG
.         WRITC     Please enter either YES or NO.
ENDQ
$END

UNIT      status
ERASE
OPEN      "class.dat",0,read,INDEXED,student_data,NONE,last_name,password
WRITE     What is your class password?
SET       ECHO,OFF
INPUT     *
GET       0,1,EQ,RESPONSE
PAUSE     ELAPSED,3
ERASE
```

```
AT      510
WRITE   Your current GPA is <<s,GPA>>.

        You currently rank number <<s,class_rank>> in your class.

        Your test scores are:
                Test #1 – <<s,test_score[1]>>
                Test #2 – <<s,test_score[2]>>
                Test #3 – <<s,test_score[3]>>
                Test #4 – <<s,test_score[4]>>
IF      meeting
.       WRITC   Please schedule a meeting with your teacher.
ENDIF
PAUSE
ENDLESSON
```

Writing Records to an Indexed File

The syntax of the PUT instruction used to write records to an indexed file is:

PUT (TAB)/(SP) channel_no

Where:

channel_no is the identifying number given to the indexed file by the OPEN instruction that accessed it.

With indexed files, the PUT instruction needs only the channel number of an indexed file as its argument. The name of the record structure containing the data to be written is specified by the OPEN instruction that opened the indexed file.

The DAL code below displays how records can be written to indexed files.

```
LESSON  add_student
DEFINE  student_data:RECORD
.       DEFINE  last_name       :STRING,25
.       DEFINE  password        :STRING,5
.       DEFINE  class_rank      :INTEGER
.       DEFINE  GPA             :REAL
.       DEFINE  test_score[4]   :INTEGER
.       DEFINE  meeting         :BOOLEAN
ENDRECORD
DEFINE  done    :BOOLEAN
OPEN    "class.dat",0,WRITE,INDEXED,student_data,NONE,last_name, password
```

```
LOOP    NOT done
.       WRITE   New student's last name?
.       INPUT
.       ASSIGN  last_name: = RESPONSE
.       WRITC   Student's password?
.       INPUT
.       ASSIGN  password: = RESPONSE
.       PUT     0                              $$ Save the info
.       WRITC   Do you have another student name to add? (Y/N)
.       INPUT
.       ASSIGN  done: = (RESPONSE<>"Y") AND (RESPONSE<>"y")
.       IF      NOT done
.       .       ERASE
.       ENDIF
ENDLOOP
ENDLESSON
```

Updating Records in an Indexed File

The UPDATE instruction modifies an existing record in an indexed file. Like the PUT instruction, UPDATE writes data from the record structure specified by the OPEN instruction that accessed the indexed file. The syntax of the UPDATE instruction is:

```
UPDATE (TAB)/(SP) channel_no
```

Where:

> *channel_no* is the identifying number given to the indexed file by the OPEN instruction that accessed it.

The UPDATE instruction replaces the contents of the current record in an indexed file with the contents of the record structure. Because UPDATE can only modify the current record in the file, the FIND instruction must be used to position the file at the record you want to modify. The syntax of the FIND instruction used with indexed files is:

```
FIND (TAB)/(SP) channel_no{, key_number, { EQ | GT | GE } key_value}
```

Where:

> *channel_no* specifies the identifying number given to the indexed file by the OPEN instruction that accessed it.

{key_number} specifies a number that specifies which key index to use to locate the record (Primary key = 0, first secondary key = 1, nth secondary key = n).

{ *EQ* | *GT* | *GE* } specifies the type of key value match you require.

{key_value} specifies the entry in the key that is also a data field in the record you need.

A FIND instruction with only a channel number as its argument makes the record listed next in the file's index the current record.

UPDATE applies only to records in indexed files.

The DAL code below displays how records can be updated in an indexed file.

```
LESSON    update
DEFINE    target      : STRING
DEFINE    option      : INTEGER
DEFINE    blanks      = "        "
DEFINE    student_data        : RECORD
.         DEFINE   last_name          : STRING,25
.         DEFINE   password           : STRING,5
.         DEFINE   major              : STRING,10
.         DEFINE   class_rank         : INTEGER
.         DEFINE   GPA                : REAL
.         DEFINE   passing            : BOOLEAN
ENDRECORD
FCOLOR    CYAN
OPEN      "class.dat",0,UPDATE,INDEXED,student_data,NONE,last_name ,password
PROMPT    ">>"
MODE      REPLACE             $$ Overtype
! Set up the screen.
AT        510
WRITE     Option :
                 1 -Change student's name/password/major
                 2 -Change student's grade information
                 3 -Quit

          Student Name:
```

```
LOOP     TRUE
.        ERASE    519;625             $$ Clear name and option fields.
.        ERASE    1024;1150
.        INPUT    519                 $$ Get option.
                  option := INT(NUMBER(RESPONSE))
.        LOOP     (option < 1) OR (option > 3)          $$ Accept only 1-3
.        .        ERASE    519;625
.        .        INPUT    519
.        .        ASSIGN   option := INT(NUMBER(RESPONSE))
.        ENDLOOP
OUTLOOP  option = 3                   $$ QUIT
.        INPUT    1024                $$ Get student name
.        ASSIGN   target := RESPONSE
.        FIND     0,1,EQ,target
.        IF       IORESULT <> 1       $$ Cannot find record.
.        .        AT       2301
.        .        WRITE    No such student. Press RETURN to continue
.        .        PAUSE
.        .        ERASE    1024;1160
.        .        ERASE    2301;2450
.        RELOOP   TRUE                $$ Start Over.
.        ELSE
.        .        GET      0,1,EQ,target
.        ENDIF
.        IF       option = 1
.        .        DO       name_change
.        ELSE
.        .        DO       grades_change
.        ENDIF
ENDLOOP
CLOSE    0
;
UNIT     name_change
AT       1210
WRITE    Press RETURN to leave fields unchanged.
         Overtype the fields you want to change.

              Last Name    : <<s,last_name>>
              Password     : <<s,password>>
              Major        : <<s,major>>

INPUT    1526                 $$ Change name
IF       RESPONSE <> ""
.        WRITE    <<s,blanks>>        $$ Erase left-over chars.
.        ASSIGN   last_name := RESPONSE
ENDIF
INPUT    1626                 $$ Change password
```

```
IF          RESPONSE <> ""
.           WRITE    <<s,blanks>>       $$ Erase left-over chars.
.           ASSIGN   password := RESPONSE
ENDIF
INPUT       1726                $$ Change major
IF          RESPONSE <> ""
.           WRITE    <<s,blanks>>       $$ Erase left-over chars.
.           ASSIGN   major := RESPONSE
ENDIF
; Save the change. TARGET still contains the original name.
FIND        0,1,EQ,target
UPDATE      0
AT          2340
WRITE       Press RETURN to continue
PAUSE
ERASE       1210;2467
;
UNIT        grades_change
AT          1210
WRITE       Press RETURN for the fields you want unchanged.
            Overtype the fields you want to change.

            GPA                 : <<t,GPA,4,2>>
            Class Rank          : <<s,class_rank>>
            Passing             : <<s,passing>> (1-PASS, 0-FAIL)

INPUT       1526                $$ Change GPA
IF          RESPONSE <> ""
.           WRITE    <<s,blanks>>       $$ Erase left-over chars.
.           ASSIGN   GPA := NUMBER(RESPONSE)
ENDIF
INPUT       1626                $$ Change password
IF          RESPONSE <> ""
.           WRITE    <<s,blanks>>       $$ Erase left-over chars.
.           ASSIGN   class_rank := INT(NUMBER(RESPONSE))
ENDIF
INPUT       1726      .         $$ Change class rank
IF          RESPONSE <> ""
.           ASSIGN   passing := INT(NUMBER(RESPONSE))
ENDIF
; Save the change. TARGET still contains the original name.
FIND        0,1,EQ,target
UPDATE      0
AT          2340
WRITE       Press RETURN to continue
PAUSE
ERASE       1210;2467
ENDLESSON
```

Deleting Records in an Indexed File

The DELETE instruction erases a record from an indexed file. Like the UPDATE instruction, DELETE is used only with indexed files and affects only the current record in a file. The syntax of the DELETE instruction is:

DELETE (TAB)/(SP) channel_no

Where:

channel_no identifies the indexed file that contains the record to be deleted.

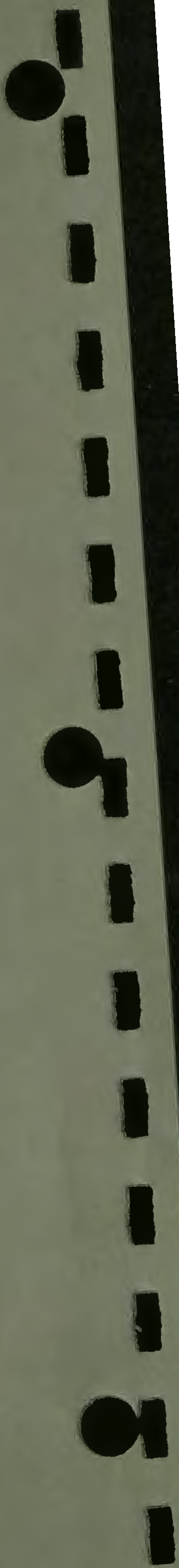
If a DELETE is executed without first positioning the file at a target record, the current record in the file is erased. Position the file at a target record with the FIND instruction. The syntax of the FIND instruction used with indexed files is listed in the section that immediately precedes this section in this chapter.

The DAL code below displays an application of the DELETE instruction to an indexed file.

```
DEFINE   target : string
AT       510
WRITE    What is the last name of the student
         whose entry you want erased?
INPUT    515
ASSIGN   target := RESPONSE
OPEN     "class.dat",0,update,INDEXED,student_data,NONE,last_name
FIND     0,1,EQ,target
DELETE   0
```

14

Parameter Passing in VAX DAL



14

Parameter Passing in VAX DAL

VAX DAL lessons can invoke VAX/VMS run-time library (RTL) routines, VAX/VMS system services, and routines written in other VAX programming languages. Because these external routines frequently need data from the main lesson, VAX DAL is equipped with mechanisms that can pass data to external routines. A VAX DAL lesson can also receive data from external routines, but only if the data is passed in the manner DAL expects.

This chapter discusses procedures for passing data into and out of DAL routines.

When developing a DAL lesson that uses both DAL routines and routines written in other VAX languages, use DAL instructions in the lesson to call all external routines. A VAX DAL lesson sets the terminal to graphics mode, whereas most other VAX language programs set the terminal to text mode. Because of this difference, attempts to call external routines in the language used by the external routines may produce unanticipated results.

PASSING DATA TO DAL ROUTINES

DAL routines receive data from other routines by two different parameter-passing methods: reference and descriptor. These two methods are used whether the data is passed to a DAL routine from another DAL routine, or from an external routine written in another VAX language. When passing data from an external routine into a DAL routine, use the same parameter-passing method that DAL would use to pass the data (see Table 14–1). Authors should not alter the default parameter-passing methods used to pass data to DAL routines.

Data is passed into DAL routines by the following default methods, based on data type.

Table 14–1: Default Parameter-Passing Methods

Data Type	Method for Passing Data
Boolean	By Reference
Integer	By Reference
Real	By Reference
String	By Descriptor
Array	By Descriptor
Table	By Reference
Record	Cannot be passed from one DAL routine to another DAL routine

When data is passed by reference, the address in memory of the variable containing the data is passed to the routine that requires the data. The routine then refers to the location for the data.

When data is passed by descriptor, a block of information describing the characteristics of the data (a descriptor) is passed to the routine that requires the data. The routine then uses the descriptor to locate and retrieve the data.

Record structures can only be declared at lesson or module level and are automatically accessed and modified during file input/output operations. Therefore, they cannot and do not need to be passed from one DAL routine to another.

DAL routines cannot receive parameters that have data types other than those supported by VAX DAL.

PASSING DATA FROM DAL ROUTINES

Data passed from one DAL routine to another DAL routine uses the default parameter-passing mechanism described in Table 14–1. Again, authors should not change the parameter-passing methods that DAL automatically uses to pass data between routines written in DAL.

When passing data from a DAL routine to a VAX/VMS RTL routine, VAX/VMS system service, or routine written in another programming language, authors can choose the method used to pass the data. The options include parameter passing by reference, by descriptor, and by value. Choose the parameter-passing option that best suits the requirements of the receiving routine.

When data is passed by value, the actual value of the data is passed directly to the external routine.

Use the arguments BY VALUE, BY REF, and BY DESC with the DO instruction to assign a parameter-passing method to a variable. The following DO instruction displays the syntax used for parameter passing:

```
DO        routine_name (var1 BY DESC, var2 BY REF, var3 BY VALUE)
```

Where:

routine_name is the name of the non-VAX DAL routine.

var1, *var2*, and *var3* identify the variables that contain the data to be passed.

BY DESC specifies parameter passing by descriptor.

BY REF specifies parameter passing by reference.

BY VALUE specifies parameter passing by value.

The BY VALUE, BY REF, and BY DESC arguments can also be used to pass parameters in DAL function calls. The following ASSIGN instruction displays the syntax used for parameter passing:

```
ASSIGN    result_var := func(var1 BY DESC, var2 BY REF, var3 BY VALUE)
```

Where:

result_var is the name of the variable that receives the value returned by the function. This variable must have the same data type as the value the function returns.

func is the name of the external routine.

var1, *var2*, and *var3* identify the variables containing the data to be passed.

BY DESC, *BY REF*, *BY VALUE* specify the methods used to pass data from the variables to the routine.

Not all data types can be passed by all three of the different parameter-passing methods. Table 14–2 shows the methods that can be used to pass variables of different data types. Possible methods are marked YES or DEFAULT. If one of the three methods is not specified, DAL passes data to external routines by the appropriate default parameter-passing method.

Table 14–2: Parameter-Passing Options (Passing Data To External Routines Only)

Data Type	By Value	By Reference	By Descriptor
Boolean	YES	DEFAULT	YES
Integer	YES	DEFAULT	YES
Real	YES	DEFAULT	YES
String	NO	YES	DEFAULT
Array	NO	YES	DEFAULT
Table	NO	DEFAULT	NO
Record	NO	YES	DEFAULT

Note that record structures can be passed to routines written in other languages. A record structure passed by descriptor is set up and passed as a fixed-length string. If a record structure is passed by reference, the address of the structure in memory is passed. The receiving routine then must call and interpret the data. Passing record structures by reference may allow external routines to define the same record format as that used in the record structure.

Permanent variables cannot be passed to routines written in other programming languages. If you need to pass a permanent variable, assign the value of the variable to a user-defined variable, and pass the user-defined variable.

Restart and student variables can be passed to external routines. Note, however, that these variables do not have any special characteristics when used in external routines.

PARAMETER-PASSING EXAMPLES

Example 1: DAL to BASIC routines

The DAL instructions and the BASIC commands shown below are both elements of the same DAL lesson. Compiled separately and then linked to the lesson, the BASIC subroutine can be treated as a unit within the lesson. DAL invokes the BASIC subroutine with a DO instruction that lists the name of the subroutine as its argument.

```
LESSON    DAL_to_BASIC
DEFINE    mystring:string        $$ input and output string
          leng:integer           $$ length of substring
FCOLOR    white
WRITE     Given a string S and a number N, this lesson calls a BASIC
          subroutine that takes N characters from the left of string,
          and moves them to the right side of the string.

          Example: S = "1234567890", N = 5, result = "6789012345"

                    Input the string to be changed:

; Get the string.
;
INPUT     *
ASSIGN    mystring := response
;
; Get the number of characters to move.
;
$GETLENG
WRITC
                    Input an integer less than <<s,LEN(mystring)>>
INPUT     *
ASSIGN    leng := INT(NUMBER(RESPONSE))
;
; Test to see if the number entered is too large.
;
IF        leng >= LEN(mystring)
.         WRITC <<s,leng>> is too large a number. Try again
.         BRANCH  $getleng
ENDIF
WRITC
                    Calling the BASIC routine ...

; Call a BASIC routine MOVE to do the string manipulation.
;
DO        move (leng BY REF, mystring BY DESC)

; Output result.
;
WRITC
                    Result: <<s,mystring>>
PAUSE
ENDLESSON
```

```
;;;;;;;;;;;;;;;; BASIC routine ;;;;;;;;;;;;;;;;;;;;;;;;;;;

100     sub move (integer leng, string mystring)
        mystring = right$(mystring,leng+1) + left$(mystring,leng)
999     end sub
```

DAL passes the variable "mystring" to the BASIC routine by descriptor, and the variable "leng" by reference.

Example 2: DAL to VAX/VMS Run-time Library Routine

The following lesson calls a VAX/VMS run-time library routine with a function call. In this example, the user passes a string to the routine and receives a string and an integer back. It is strongly recommended that authors treat run-time library routines as functions. In this way, the author can check the status value returned by the routine.

When strings are returned from run-time library routines, the variable that receives the string must be assigned a null string before the routine executes, as the example below illustrates.

```
LESSON    samp1
DEFINE    logname            : string       $$ Variable for the logical name
DEFINE    ln                 : integer      $$ Length of the translation
DEFINE    result             : string       $$ Translation string
DEFINE    status             : integer      $$ status value from RTL call
DEFINE    lib$sys_trnlog     : integer, function
ASSIGN    logname := "SYS$SYSTEM"
ASSIGN    result := ""                $$ Initialize translation string variable
ASSIGN    status := lib$sys_trnlog(logname BY DESC,ln BY REF,result BY DESC)
AT        310
WRITE     SYS$SYSTEM = <<s,result>>
WRITC     The translation is <<s,ln>> characters long.
WRITC     STATUS = <<s,status>>
PAUSE
ENDLESSON
```

Example 3: DAL to VAX/VMS Run-time Library Routine

The following lesson illustrates the alternative approach to passing parameters to run-time library routines. In this instance, the routine is treated as an external unit or subroutine. Note that there is no mechanism for obtaining a status value in this approach.

```
LESSON    samp2
DEFINE    logname          : string      $$ Variable for the logical name
DEFINE    ln               : integer     $$ Length of the translation
DEFINE    result           : string      $$ Translation string
ASSIGN    logname := "SYS$SYSTEM"
ASSIGN    result := ""                $$ Initialize translation string variable
DO        lib$sys_trnlog(logname BY DESC,ln BY REF,result BY DESC) $$ CALL IT
AT        310
WRITE     SYS$SYSTEM = <<s,result>>      $$ Display the result
WRITC     The translation is <<s,ln>> characters long.
PAUSE
ENDLESSON
```

Example 4: DAL to VAX/VMS System Service

The following lesson invokes a VAX/VMS system service that, given a VAX/VMS status code, returns text that explains the code. As in Example 2 above, the called routine is treated as an external function. Because the system service returns a string, the string variable that receives the string is loaded with a null string before the service is invoked.

```
LESSON    samp3
$again
AT        1000
PROMPT    "Enter message number (STOP to exit): "
INPUT
IF        RESPONSE <> "STOP"
.         DO         display_status (INT(NUMBER(RESPONSE)))
.         ERASE
.         BRANCH     $again
ENDIF
!
! Given a status code, look up the message and display it.
! Call the system service $GETMSG to get the text.
!
UNIT      display_status (status_code)
DEFINE    status_code,status,msglen : integer
DEFINE    msgtext : string
DEFINE    sys$getmsg : integer,function
!
! Initialize the msgtext variable to the maximum length of the text
! string we want to allow. Any excess past the initialized length
! will be discarded.
!
ASSIGN    msgtext := "
```

```
ASSIGN   status := sys$getmsg(status_code BY VALUE,   $$ status code of message
                  msglen BY REF,                      $$ location for length
                  msgtext,                            $$ buffer for text
                  -1 BY VALUE,                        $$ flags, return all parts
                  0 BY VALUE)                         $$ optional
ASSIGN   msgtext := SUBSTR(msgtext,1,msglen)
WRITC    <<s,msgtext>>
DO       waiter
RETURN
!
! Routine for pausing
!
UNIT     waiter
!
! Wait for them to read it
!
PROMPT   "Press RETURN to continue"
AT       2220
INPUT
RETURN
ENDLESSON
```

15

Macros in VAX DAL

VAX DAL macros offer one way to speed lesson development and simplify lesson modification. If certain operations are performed frequently in your lesson, you can use macros to save a significant amount of development time.

When macros are used, you do not need to retype a frequently used series of instructions, or call a unit that contains the series, each time the series is needed in the source code of the lesson. Instead, include the instructions in a macro and gives the instructions a tag, or *macro name*. Then, write the macro name into the source code wherever you need the instructions. At compile time, the entire series of instructions is written to the listing file at the each place you used the macro name.

Macros can be made to process arguments. The instructions the macro inserts in the listing file can be given different arguments each time the macro is used.

The instructions that create and include macros in the listing file of a lesson begin with a percent sign (%). These instructions are:

%MACRO
%ENDMACRO
%INCLUDE

This chapter describes the instructions used to create VAX DAL macros and provides examples of how macros can be used.

DEFINING A VAX DAL MACRO

The syntax of the %MACRO instruction that begins a macro definition is:

%MACRO (TAB)/(SP) macro_name {(p1, p2, ...)}

Where:

macro_name is the name given the series of instructions that is to be written into the file. A macro name can consist of one or more alphanumeric characters, and can include the punctuation marks dollar sign ($), underscore(_), and period (.). Macro names must begin with an alphabetic character.

{(p1, p2, ...)} is an optional list of one or more parameters, or arguments, to the macro. Up to 255 parameters can be used. Parameters must not be enclosed in quotation marks unless the quotation marks are part of the parameter. A parameter name can consist of one or more alphanumeric characters, and can include the punctuation marks dollar sign ($), underscore (_), and period (.). Parameters cannot include commas or parentheses unless the parameters are enclosed in quotes.

A macro definition begins with the %MACRO instruction, includes the instructions that are to be written into the listing file, and ends with the %ENDMACRO instruction. The %MACRO and %ENDMACRO instructions cannot follow dot indentation; they must begin in the leftmost column in the source file. The DAL source code below shows a simple macro definition that writes the "press RETURN" message into the listing file.

```
%MACRO  press_ret
AT      2255
MODE    INVERSE
WRITE   ... Press RETURN to continue ...
MODE    NORMAL
PAUSE
%ENDMACRO
```

After the above macro definition, enter the word "press_ret" in the source code wherever you need the lesson to pause. At compile time, all of the instructions in the macro are included in the listing file at each place press_ret appears.

Define a macro before you use the name of the macro in the lesson. VAX DAL does not support forward references to macro definitions.

USING PARAMETERS WITH A VAX DAL MACRO

Parameters can expand the usefulness of a macro. A macro with parameters writes the same set of instructions into the listing file each time the macro is used, but assigns the instructions different arguments.

List parameters after the macro name in the order in which they are used in the macro. Parameters must be enclosed in single quotation marks in the body of the macro definition. The following macro definition creates a macro that opens an indexed file, assigns the file a channel number, and specifies the name of the file's primary key. Next, the macro specifies the key value of a record to be retrieved from the file.

```
%MACRO  open_it (name, chan, pkey, get)
OPEN    'name', 'chan', read, INDEXED, data_rec, none, 'pkey'
GET     'chan', 0, EQ, 'get'
CLOSE   'chan'
%ENDMACRO
```

Rather than typing the OPEN, GET, and CLOSE instructions and all of their arguments to retrieve a record, the author can retrieve a record with the words "open_it" and four parameters that specify the file and the record needed. A use of the open_it macro and a listing of the source code the macro produces is shown below.

In the source code:

```
open_it    ("file.dat", 3, last_name, Smith)
```

In the listing file:

```
OPEN    "file.dat", 3, read, INDEXED, data_rec, none, last_name
GET     3, 0, EQ, Smith
CLOSE   3
```

Used with different parameters, the open_it macro can generate instructions that open and retrieve data from any number of different files.

If the number of parameters listed after the macro name in the source code is less than the number of parameters required by the instructions in the macro, null strings are written in the listing file in place of the missing parameters. If too many parameters are listed after the macro name, the extra parameters are discarded.

No data type checking occurs with macro parameters. Parameters are written to the listing file purely by text substitution.

Macro definitions can be nested. That is, one macro can be defined inside another, and the internal macro can use the parameters assigned the larger macro of which it is a part. The macro definitions below are nested. The source code file is:

```
LESSON    Tmp
%MACRO    show_it (picfile, fkey)
SLIDE     'picfile'
%MACRO    open_it (chan)
OPEN      'picfile', 'chan', UPDATE, SEQUENTIAL
WRITE     Now opening file: 'picfile' on channel 'chan'
%ENDMACRO
PAUSE     STRING, 'fkey'
%ENDMACRO

show_it   ("j1.pic", "[F10_KEY]")
open_it   (6)
ENDLESSON
```

The resulting listing file is:

```
LESSON    Tmp
SLIDE     "j1.pic"
PAUSE     STRING, "[F10_KEY]"
OPEN      "j1.pic", 6, UPDATE, SEQUENTIAL
WRITE     Now opening file: "j1.pic" on channel 6
ENDLESSON
```

USING THE %INCLUDE INSTRUCTION

Because of the way the DAL compiler processes macros and secondary DAL source files, authors may need to use the %INCLUDE instruction in lessons that contain macros and consist of two or more files.

The VAX DAL compiler preprocesses macros before it parses a source file. Parsing removes any unnecessary spaces from the files. If a second file is included in the main lesson file at compile time by an INCLUDE instruction, the second file is parsed but not preprocessed. Consequently, any macros in the included file are not compiled.

The %INCLUDE instruction includes a secondary file in preprocessing and ensures that macros in the file are properly compiled. If your lesson consists of two or more DAL source files, one of which includes the other file at compile time, and if the included file contains macros, use %INCLUDE instead of INCLUDE to join the files at compile time. The syntax of the %INCLUDE instruction is:

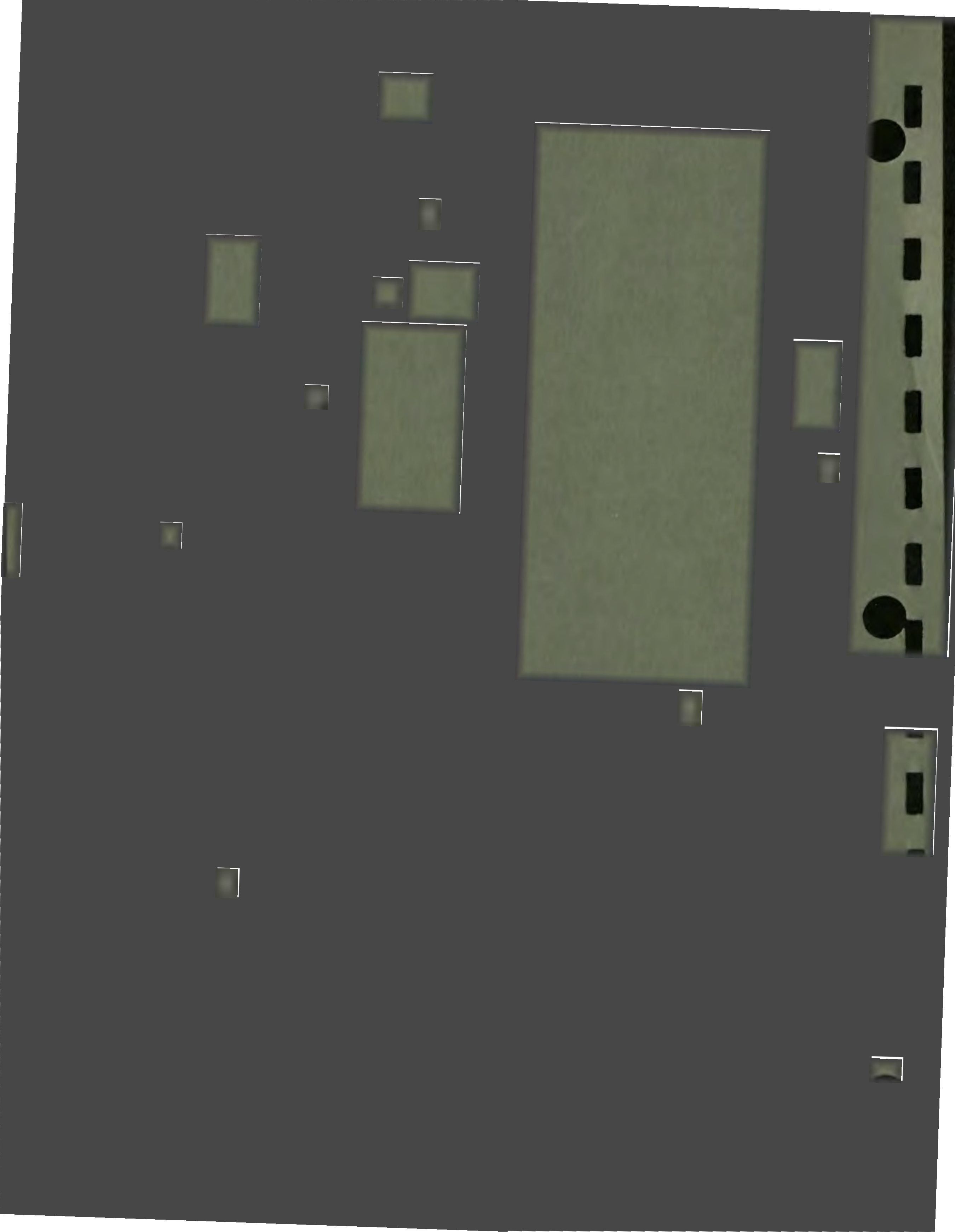
%INCLUDE (TAB)/(SP) file_specification

Where:

file_specification is the file name and extension of the file to be included. This specification is not enclosed in quotation marks.

The %INCLUDE instruction must begin in the leftmost column of the main DAL lesson file. A file included in compilation by an %INCLUDE instruction cannot contain %INCLUDE instructions.

A
Instructions



A
Instructions

This appendix gives a brief description of all instructions in DAL. The instructions that are explained in this manual are preceded by an asterisk. The *VAX DAL Reference Manual* gives a full explanation of all VAX DAL instructions.

***ASSIGN** — The ASSIGN instruction assigns a value to a variable.

***AT** — The AT instruction selects a screen address.

AXES — The AXES instruction defines the screen location of the maximum boundaries of the axes of a graph.

***BACKUP** — The BACKUP instruction transfers control to another unit in the current unit calling chain, or transfers control back to lesson level.

***BCOLOR** — The BCOLOR instruction modifies the color of the screen. When a black-and-white monitor is used, the colors specify shades of gray from darkest to lightest.

***BOX** — The BOX instruction draws a rectangle on the screen.

***BRANCH** — The BRANCH instruction transfers control to another point in the lesson. The transfer is conditional, depending on the arguments to BRANCH. Unconditional transfer is allowed within the current unit.

***CANCEL** — The CANCEL instruction disables a condition handler. CANCEL only applies to condition handlers declared at the current level of the lesson.

***CCOLOR** — The CCOLOR instruction clears pointers from the DAL color table to allow for new color selections.

***CDUNIT** The CDUNIT instruction opens and names a DAL unit that supports a condition handler.

CHAR The CHAR instruction modifies one character in an alternate character set. The CHAR instruction specifies the dot pattern displayed on the screen.

CHARSET The CHARSET instruction specifies one of the four character sets as active. The dot patterns in this character set are used in writing text until another CHARSET instruction selects another character set. The standard ASCII character set is one of the four; the other three are user defined.

***CHECKERR** The CHECKERR instruction begins a block of instructions that checks the value of the ERRORV system variable in a QUERY block.

***CIRCLE** The CIRCLE instruction draws a circle whose center and radius are specified. When optional arguments are used to specify the beginning and end of an arc, CIRCLE draws only the arc.

CLIP The CLIP instruction defines a rectangular area on the screen for displays. After this area is defined, only those locations within the area are displayed, regardless of the addresses used with WRITE and the various graphics instructions.

CLOSE The CLOSE instruction closes a file so that its contents are no longer available to the lesson.

CONTROL The CONTROL instruction sends to the terminal control sequences that are not ReGIS commands.

***CONVERT** The CONVERT instruction specifies the formula used to convert a value expressed in one unit of measure to an equivalent value in another unit. The student can use either unit in responses.

***CURVE** The CURVE instruction fits a curve to a specified number of points on the screen.

DECTALK The DECTALK instructions specify how a DECtalk unit is connected to a terminal, and what characteristics the DECtalk unit is to have.

***DEFINE**	The DEFINE instruction defines the name, data type, and usage characteristics of variables, and the name and value of constants.
***DELETE**	The DELETE instruction deletes the current record in an indexed file.
DELIMIT	The DELIMIT instruction specifies the character that ends a student's response.
DELTA	The DELTA instruction specifies a standard increment for the independent variable in a GRAPH instruction.
***DO**	The DO instruction executes the unit or the lesson named as its argument. This instruction can be used at lesson level to control the sequence of units in the entire lesson, and at unit level to control execution of one unit from another.
DO *	The DO * instruction calls another program or VAX/VMS software utility. Authors can invoke DAL lessons or programs written in other languages.
***DOT**	The DOT instruction illuminates the smallest addressable screen element.
***ELSE**	The ELSE instruction marks the beginning of a series of instructions that are executed only when the condition specified by the preceding IF instruction is false.
ENDFILL	The ENDFILL instruction marks the end of a block of graphics instructions that began with the FILL instruction.
***ENDFOR**	The ENDFOR instruction marks the end of the iterative instructions begun by the preceding FOR instruction.
***ENDIF**	The ENDIF instruction marks the end of an IF structure.
***ENDLESSON**	The ENDLESSON instruction is the last instruction in a VAX DAL source file that contains lesson-level instructions.
***ENDLOOP**	The ENDLOOP instruction marks the end of the instructions that are repeated as long as the condition specified by the preceding LOOP instruction is true.
***ENDMODULE**	The ENDMODULE instruction is the last instruction in a VAX DAL source file that does not contain lesson-level instructions.

***ENDQ** The ENDQ instruction marks the end of a series of response-judging instructions begun by the QUERY instruction.

***ENDRECORD** The ENDRECORD instruction specifies the end of a record structure definition for the compiler. Record structures are used in many file I/O operations.

***ENDTEST** The ENDTEST instruction marks the end of a TEST structure.

***ERASE** The ERASE instruction erases all or part of the screen, refreshing it with background color.

***FCOLOR** The FCOLOR instruction specifies a color to be used in text and graphics displays.

FILL The FILL instruction begins a block of graphics instructions that draws a polygon. The interior of the polygon is shaded with the current foreground color and pattern.

***FIND** The FIND instruction locates a record in an indexed or random-access file.

***FOR** The FOR instruction begins an iterative structure and specifies the number of times the instructions between the FOR and its associated ENDFOR will be executed.

FUNCT The FUNCT instruction begins a specialized subroutine that either defines a new function which then can be called like the system functions, or redefines an existing system function.

GAT The GAT instruction is the graphing analog of the AT instruction. The GAT instruction selects a screen address using the current GORIGIN and the current scale to locate the point specified by GAT on the screen.

GBOX The GBOX instruction is the analog of BOX using the graphing coordinates.

GCIRCLE The GCIRCLE instruction is the graphing analog of CIRCLE.

GCURVE The GCURVE instruction is the graphing analog of CURVE.

GDOT The GDOT instruction is the graphing analog of DOT.

***GET**	The GET instruction reads a record from an open file into a variable or record structure.
GLINE	The GLINE instruction is the graphing analog of LINE.
***GOAL**	The GOAL instruction divides units of a lesson into groups for which separate scores are accumulated.
GORIGIN	GORIGIN is one of the instructions that define the screen location and the scale for graphs. GORIGIN defines the point on the screen that is the origin of the graphing system. AXES defines the screen location of the ends of both axes. SCALEX and SCALEY define the scale.
GRAPH	The GRAPH instruction graphs the expression used as its argument. The origin and scale of the graph are established by the related instructions GORIGIN, AXES, SCALEX and SCALEY, and LSCALEX and LSCALEY. The independent variable in the expression is incremented by the instruction DELTA.
GVECTOR	The GVECTOR instruction is the graphing analog of VECTOR.
HBAR	The HBAR instruction draws one horizontal bar of a bar graph. The current graph origin, dimensions, and scale — defined by the most recent GORIGIN, AXES, and SCALEX and SCALEY instructions — determine the placement of the bar on the screen.
***IF**	The IF instruction marks the beginning of an IF structure and specifies the condition to be tested.
INCLUDE	The INCLUDE instruction inserts the VAX DAL source code in another file into this source file at compile time.
***INPUT**	The INPUT instruction displays the prompt character and waits for a response from the keyboard.
***ITALICS**	The ITALICS instruction specifies that subsequent text is to be italicized.
***JUDGE**	The JUDGE instruction modifies the judgment of student responses. Both the judgment that a response is right or wrong and the events that follow the judgment can be changed.

***LESSON**	The LESSON instruction begins a lesson and specifies the lesson name.
***LINE**	The LINE instruction draws a straight line from one point on the screen to another.
LOG	The LOG instruction creates a file containing information about student performance. As different students take the lesson, the file is updated.
***LOOP**	The LOOP instruction marks the beginning of a LOOP structure. A LOOP structure contains a series of instructions that are executed as long as the condition specified by LOOP is true.
LSCALEX, LSCALEY	The LSCALEX and LSCALEY instructions specify logarithmic scaling factors used to map graphs to the screen. With GORIGIN and AXES, LSCALEX and LSCALEY provide the basis for defining the graphing coordinate system with logarithmic scales.
***MAP**	The MAP instruction loads color specifications and gray levels into DAL's internal color map.
***MARKUP**	The MARKUP instruction writes the contents of the system variables OKWORD or NOWORD on the screen.
MARKX, MARKY	The MARKX and MARKY instructions draw the axes of a graph at the screen locations defined by GORIGIN and AXES. The markings, both tic marks and numbers, are selected by the arguments to MARKX and MARKY.
MATCH	The MATCH instruction determines the presence of a particular word in the student's response.
MGRAPH, ENDMGRAPH	The MGRAPH/ENDMGRAPH instruction pair specifies a macrograph letter and defines the ReGIS commands it is associated with.
MLOAD	The MLOAD instruction loads a macrograph into the terminal.
***MODE**	The MODE instruction specifies a mode of graphics display that affects the appearance of text and graphics on the screen and the way display information is stored in the terminal's bit-map memory.

***MODULE**	The MODULE instruction begins a separately compiled module and specifies the module's name.
MPLOT	The MPLOT instruction draws a macrograph loaded into the terminal by a previous MLOAD instruction.
***NOISE**	The NOISE instruction specifies words to be ignored when the student uses them in a response.
NOWORD	The NOWORD instruction specifies the word displayed when the instruction MARKUP is used as a response-contingent instruction for an answer judged wrong.
OKWORD	The OKWORD instruction specifies the word displayed when the MARKUP instruction is used as a response-contingent instruction with an answer judged OK.
***ON**	The ON instruction establishes a condition handler.
***OPEN**	The OPEN instruction opens a file and establishes both the channel number used to identify the file and an access mode for the file.
***OTHER**	The OTHER instruction marks the beginning of a series of instructions that are executed only if none of the VALUE instructions between the preceding TEST instruction and the OTHER instruction have caused a different set of instructions to be executed.
OUTLOOP	The LOOP instruction specifies a condition that must be false if control is to be passed to the instruction following the ENDLOOP. The OUTLOOP instruction specifies an alternate condition that causes control to pass to the instruction following the ENDLOOP.
***PATTERN**	The PATTERN instruction selects a pattern for line drawings and for shading the area specified by SREF.
***PAUSE**	The PAUSE instruction suspends execution of a lesson until one of the following occurs: a specified amount of time elapses, a specified time is reached, the RETURN key is pressed, or a correct response is entered.
***PROMPT**	The PROMPT instruction defines the character displayed as the prompt by the QUERY and INPUT instructions. It can specify a prompt string from an alternate character set.
***PUT**	The PUT instruction writes a record into a file.

***QUERY**	The main function of the QUERY instruction is to mark the beginning of a response-judging block of instructions. The QUERY instruction also displays the prompt character and pauses until a response has been typed on the keyboard.
***RAT**	The RAT instruction is the analog of AT in the relative addressing system.
***RBOX**	The RBOX instruction is the analog of the BOX instruction in the relative addressing system.
***RCIRCLE**	The RCIRCLE instruction is the analog of CIRCLE in the relative addressing system.
***RCURVE**	The RCURVE instruction is the analog of CURVE in the relative addressing system.
***RDOT**	The RDOT instruction is the analog of DOT in the relative addressing system.
REDO	The REDO instruction reexecutes the current unit when it is used at unit level, or reexecutes the lesson when it is used at lesson level.
REGIS	The REGIS instruction sends an unaltered ReGIS string to the terminal.
RELOOP	The RELOOP instruction, like the OUTLOOP instruction, specifies a condition that passes control to the preceding LOOP instruction before the ENDLOOP is reached.
***RESTORE**	The RESTORE instruction replaces current terminal characteristics with characteristics previously saved by the SAVE instruction.
***RETURN**	The RETURN instruction returns control to the calling unit or lesson before the end of the unit or lesson in which it is executed.
***RIGHT**	The RIGHT instruction specifies anticipated right answers to a QUERY. The RIGHT instruction can also mark the beginning of a series of response-contingent instructions that are executed only if the student's response matches a right answer.

***RIGHTV** — The RIGHTV instruction is a version of the RIGHT instruction that specifies right answers as variables or expressions. The RIGHTV instruction is used: when the student's response is an expression to be evaluated; when the answer has been found by evaluating expressions, as is often the case in mathematics; when files are used to store sets of questions and answers; and in other cases when the right answer is more easily specified as a variable than as a constant.

***RLINE** — The RLINE instruction is the relative graphics analog of LINE.

***RORIGIN** — The RORIGIN instruction defines the origin for relative graphics. The addresses specified in the arguments to R-prefixed graphics instructions are interpreted as relative to the point defined by RORIGIN.

***ROTATE** — The ROTATE instruction specifies that the addresses used for subsequent relative graphics (R-prefix instructions) are rotated a specified number of degrees before the figure is drawn.

***RSIZE** — The RSIZE instruction modifies the vertical and horizontal dimensions of relocatable graphics.

***RVECTOR** — The RVECTOR instruction is the relative graphics analog of the VECTOR instruction.

***SAVE** — The SAVE instruction creates a restorable set of terminal characteristics. Sets of characteristics are restored to the terminal by the RESTORE instruction.

SCALEX, SCALEY — The SCALEX and SCALEY instructions specify the scaling factors used to map graphs to the screen.

***SCORE** — The SCORE instruction enables and disables scoring. The SCORE instruction can be used at any point in the lesson or in a unit. Scoring is on by default.

***SCORE UPDATE** — The SCORE UPDATE instruction updates scoring system variables in a unit after response judging, but before execution of the unit finishes. By default, scoring variables update only after a unit finishes.

SEED — The SEED instruction seeds random number algorithms with the current value of the clock to generate a new sequence using the RANDOMx functions.

***SET ECHO**	The SET ECHO instruction specifies whether student responses are echoed on the screen.
***SET FKEY**	The SET FKEY instruction specifies whether students can enter special function keys in response to INPUT, QUERY, or PAUSE STRING instructions.
***SET HLS**	The SET HLS instruction indicates whether the terminal in use supports the HLS (Hue/Lightness/Saturation) method for specifying colors.
***SET KEYPAD**	The SET KEYPAD instruction selects the numeric or application mode for the terminal keypad. In the application mode, keypad keys generate ASCII strings that can be treated as responses to INPUT, QUERY, or PAUSE STRING instructions.
***SET MAXCOLORS**	The SET MAXCOLORS instruction specifies the number of colors that can be displayed on a terminal screen simultaneously.
***SET TYPEAHEAD**	The SET TYPEAHEAD instruction permits or prevents students from typing answers ahead.
***SIGNAL**	The SIGNAL instruction invokes a CONDITION,value condition handler.
***SIZE**	The SIZE instruction alters the size of the text.
SLIDE	The SLIDE instruction displays a filename.PIC file containing a slide.
SPEAK	The SPEAK instruction sends text to a DECtalk unit to be spoken.
***SPECS**	The SPECS instruction modifies the requirements for judging a student right or wrong by changing the default specifications for exactness of spelling, punctuation, and capitalization, for allowing extra words or the same words in different order, and for allowing expressions or requiring units of measure in the response. The SPECS instruction can also make unit conversion formulas and system functions unavailable for use in student responses.

***SREF**	The SREF instruction specifies a reference line or point used to shade figures drawn with the graphics instructions. The area from the reference line or point to the line drawn by subsequent graphics instructions is shaded in the current foreground color and the current pattern. SREF can also specify a character pattern to be used for shading.
STOP	The STOP instruction ends execution of the lesson. Restart variables are saved when the lesson ends with STOP.
***SYN**	The SYN instruction specifies synonyms for use in answer processing by identifying a target word and a list of synonyms. When a target word is specified within a RIGHT or WRONG instruction, all of the synonyms in the list are considered equivalent in judging the student's response.
***TEST**	The TEST instruction marks the beginning of a test structure that ends with the instruction ENDTEST and specifies the name of the variable that is the basis for the test.
TRAY	The TRAY instruction loads a tray file created by the Slide Projection System and makes it available for subsequent SLIDE instructions.
***TROTATE**	The TROTATE instruction specifies that subsequent text will be rotated a multiple of 45 degrees from horizontal, depending on the value specified.
***UNIT**	The UNIT instruction specifies the unit's name in the form needed by the DO instruction and identifies the beginning of the unit.
***UPDATE**	The UPDATE instruction writes the contents of the record structure into the current record in an indexed file.
***VALUE**	The VALUE instruction is part of a TEST structure. The VALUE instruction specifies a value, a series of values, or a range of values to be compared to the contents of the variable specified in its associated TEST instruction, and marks the beginning of a series of instructions that are executed only if the values in its argument match the value of the variable.
VBAR	The VBAR instruction draws one vertical bar of a bar graph.
***VECTOR**	The VECTOR instruction draws an arrow between any two points with an arrow head of an arbitrary size.

WEIGHT The WEIGHT instruction defines the value added to the SCORE and SCORES(i) variables when the student responds to a QUERY.

WHEN The WHEN instruction specifies that characters input from the keyboard, elapsed time, or absolute time can interrupt the current unit and cause execution of another unit.

***WRITC** The WRITC instruction is a variation of the WRITE instruction. The only difference is that DAL inserts a carriage return before the first line of text. Alternatively, this instruction can be spelled WRITEC.

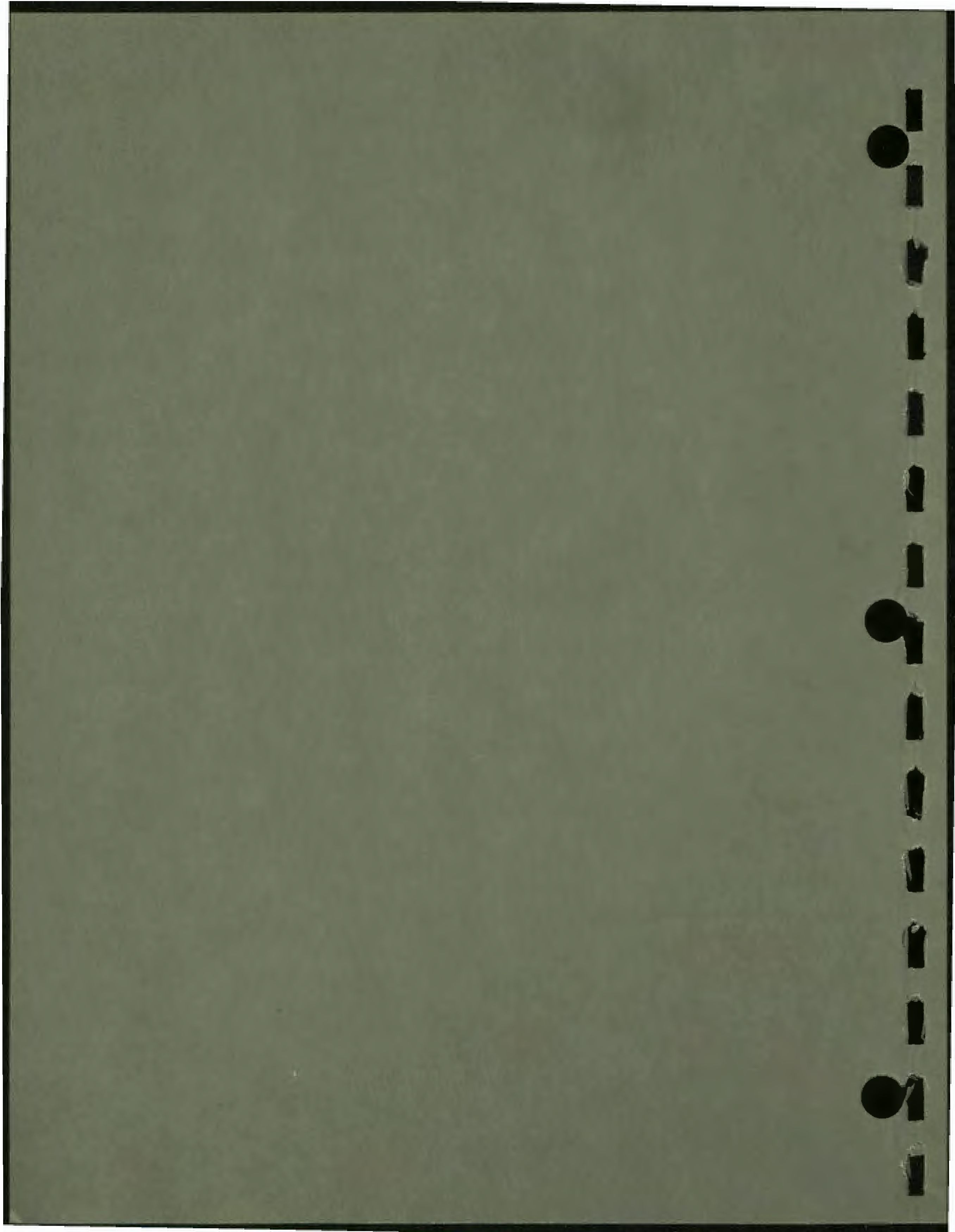
***WRITE** The WRITE instruction specifies the text and variables whose contents are to be displayed on the screen.

***WRONG** The WRONG instruction specifies anticipated wrong answers to a QUERY. The WRONG instruction can also mark the beginning of a series of response-contingent instructions that are executed only if the student's response matches this answer.

***WRONGV** The WRONGV instruction is a version of the WRONG instruction that specifies wrong answers as variables or expressions. The WRONGV instruction is used: when the student's response is an expression to be evaluated; when the answer has been found by evaluating expressions, as is often the case in mathematics; when files are used to store sets of questions and answers; and in other cases when the wrong answer is more easily specified as a variable than as a constant.

B

System Functions



B

System Functions

This appendix lists all system functions. The functions explained in this manual are preceded by an asterisk.

ABS(x) returns the absolute value of its argument, which can be a scalar or an array.

ALT(a,b,...) inserts a separator (a CTRL/P) into a string between the part of the string expressed as argument a and the part of the string expressed as argument b. A string can be divided into any number of fields, with each field delimited by a CTRL/P.

ANTILOG(x) returns the antilogarithm of its argument.

ARCCOS(x) returns the angle whose cosine is its argument. The angle is returned as a decimal in radians.

ARCSIN(x) returns the angle whose sine is its argument. The angle is returned as a decimal in radians.

ARCTAN(x) returns the angle whose tangent is its argument. The angle is returned as a decimal in radians.

ARCTAND(x,y) returns the angle whose tangent is x/y. If y evaluates to 0, pi/2 is returned in radians.

ASCII(x,i) requires a string constant or variable in x and returns the ASCII value of the character in position i in the string.

CHAR(x) returns the character that has an ASCII value of x.

CHARSETLD(x) returns the Boolean value 1 if the character set x has been loaded during this execution of the lesson. The argument x must be identical to the argument to the CHARSET instruction.

COS(x) returns the cosine of its argument. The argument must be in radians.

DEG(x) returns the value of its argument in degrees. The argument must be in radians.

DELETE(x,y) returns a subset of string x with y deleted. The string y can be a pattern-valued expression. Only the first instance of y is deleted even if y appears more than once. If the substring y is not in the string x, x is returned unchanged.

DET(array) returns the determinant of the square array given as its argument.

DIMS(array) returns the number of dimensions of the array named.

DIMAX(array,i) returns the upper bound of the ith dimension of the array named.

***EOF(channel no)** returns the Boolean value 1 if the file associated with the channel is at its end-of-file mark. The EOF function returns the Boolean value 0 if the file is not at its end-of-file mark. This function is used with the PUT instruction. This function sets the system variable IORESULT.

EVAL(string) takes a character string as its argument, evaluates the string as an expression, and returns a real number result. DAL operators and the following system functions can be used within the EVAL function.

ABS	ANTILOG	ARCCOS
ARCSIN	ARCTAN	ARCTAND
COS	DEG	EXP
INT	LN	LOG
RAD	REAL	ROUND
SIN	SQRT	TAN

EXP(x) returns a value equal to e raised to the power of the argument.

***FIND(channel no,record no)**	attempts to position the file associated with the channel number at the specified record number, and returns the Boolean value 1 if the attempt is successful. If the record number is lower than the record number at the current position, REWIND occurs automatically. This function sets the system variable IORESULT. The FIND function cannot be used with indexed files.
IDEN(x)	returns an identity array of order x which can be used directly in expressions, or can be assigned to a variable.
INSTRING(x,y)	returns a number corresponding to the character position of string y in string x. If string y contains more than one character, the position of the first character is returned. If y is not contained in x, 0 is returned.
***INT(x)**	returns the truncated integer portion of its real number argument. The sign is preserved.
INV(array)	returns the inverse of its argument which must be a square array.
IS	changes the three conversion functions NUMBER, INT, and INV to interrogative functions. ISNUMBER(x) returns 1 if x is either a real number or an integer, and 0 otherwise. ISINT(x) returns 1 if x is an integer. ISINV(x) returns 1 if x has a nonzero determinant.
LEN(x)	returns the length (number of characters) of the string variable or record structure listed as its argument. An integer value is returned.
LN(x)	returns the logarithm (base e) of its argument.
LOG(x)	returns the logarithm (base 10) of its argument.
LOWER(x)	returns a string with all uppercase letters in its string-valued argument converted to lowercase (digits and symbols are not modified).
***NUMBER(x)**	returns the real number represented by its string-valued argument.

PICFILE (filename)	opens the specified file and sends the ReGIS instructions for all subsequent screen displays to this file as well as to the screen. The file is closed by the function with no argument. The file extension can be included; the default is .PIC. The PICFILE function returns a Boolean 1 if the file was successfully opened or closed.
RAD(x)	returns the value of its argument in radians. The argument is assumed to be in degrees and is evaluated mod 360.
RANDOMN (m,sd)	returns a randomly selected number given a mean and a standard deviation of a normal distribution.
RANDOMP (lambda)	returns a randomly selected number between 0 and 1 selected from a Poisson distribution of intensity lambda.
***RANDOMU(x,y)**	returns a randomly selected number between x and y drawn from a uniform distribution. The value of x is included in the distribution. The value of y is not.
***REAL(x)**	returns a real number equivalent to its integer argument.
REPLACE(x,y,z)	returns string x with the first instance of string y replaced by string z. Replacement occurs from left to right. If string y does not exist in string x, string x is returned unchanged.
***REWIND (channel no)**	resets the file associated with the specified channel number to its first record. Returns 1 if the rewind is successful; returns 0 if it is not. This function sets the system variable IORESULT.
ROUND(x)	returns an integer that is equivalent to the rounded real number argument. The sign is preserved.
SIGN(x)	returns +1 if the value of x is a positive number and −1 if the value of x is a negative number. If the value of x is 0, 0 is returned. The argument can be a real number, an integer, or an array.
SIN(x)	returns the sine of its argument. The argument must be in radians.
SQRT(x)	returns the square root of its argument. The argument must be positive, and cannot be a table or an array.
STRING(x)	returns a character string equivalent to its real or integer argument.

SUBSTR (string,y,z) returns the substring of string that begins at character position y and is z characters long. Returns a null string if string does not contain at least z characters starting at position y.

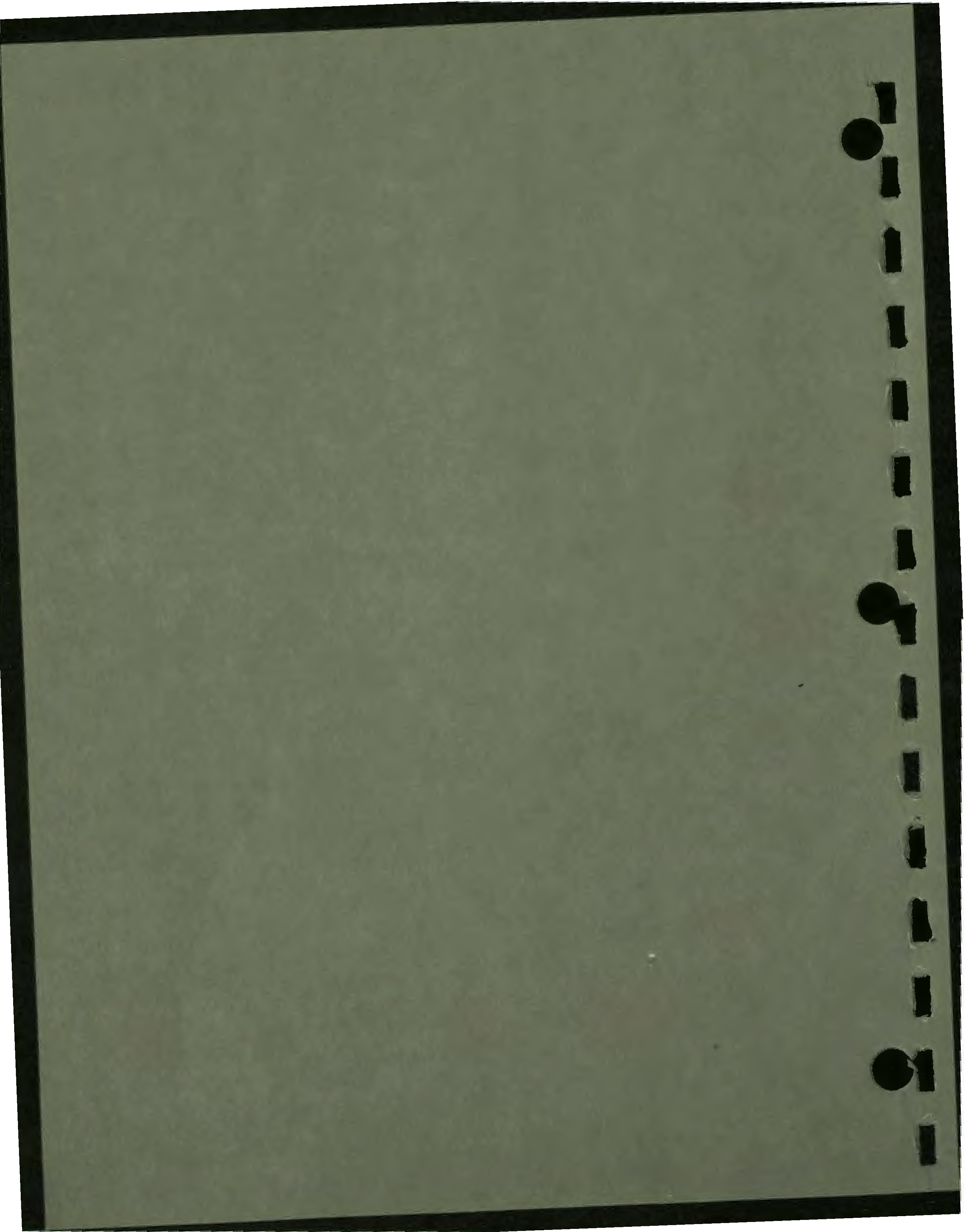
TAN(x) returns the tangent of its argument. The argument must be in radians.

TRAN(array) returns the transpose of the array specified as its argument.

UPPER(x) returns a string with all lowercase letters in its string-valued argument converted to upper case. Digits and symbols are not changed.

WORD(n,string) returns the nth word in the specified string variable. If n evaluates to less than 1 or to a number greater than the number of words in the string, a null string is returned. Words are terminated by a space, a RET, a TAB, or any punctuation mark except apostrophe, hyphen, dollar sign, and underscore. Other punctuation marks are ignored, and are not returned as part of the word.

C
System Variables



System Variables

This appendix lists the system variables. The variables are listed alphabetically within each of three categories. Table C–1 lists the graphics and graphing system variables. Table C–2 lists the scoring system variables. Table C–3 lists the response-related system variables. Table C–4 lists the timing and miscellaneous system variables.

Table C–1: Graphics and Graphing System Variables

Name	Data Type	Description	Modified By
BCOLOR	integer	Value of current background color.	BCOLOR instruction
CHARSET	string	Name of last character set specified.	CHARSET instruction
FCOLOR	integer	Value of current foreground color.	FCOLOR instruction
GORIGINX	real	Value of x coordinate last specified.	GORIGIN instruction
GORIGINY	real	Value of y coordinate last specified.	GORIGIN instruction

Table C–1: Graphics and Graphing System Variables (Cont.)

Name	Data Type	Description	Modified By
GRAFSTAT	read-only array	Information about the current state of the display. See Appendix A in the VAX DAL Reference Manual for a listing of the GRAFSTAT array elements.	Various graphics instructions.
GWHEREX	real	X coordinate of the last point displayed expressed in current graph scale. If text was displayed last, this point is the upper left corner of the character cell.	Updated by all graphics instructions. Reset by GORIGIN, SCALE, and LSCALE
GWHEREY	real	Y coordinate of the last point displayed expressed in current graph scale. If text was displayed last, this point is the upper left corner of the character cell.	Updated by all graphics instructions. Reset by GORIGIN, SCALE, and LSCALE
ITALICS	integer	Value last specified.	ITALICS instruction
RORIGINX	integer fine coordinates	X coordinate of point last specified.	RORIGIN instruction
RORIGINY	integer fine coordinates	Y coordinate of point last specified.	RORIGIN instruction
RSIZEX	integer	X size coefficient.	RSIZE instruction
RSIZEY	integer	Y size coefficient.	RSIZE instruction

Table C–1: Graphics and Graphing System Variables (Cont.)

Name	Data Type	Description	Modified By
RWHEREX	integer fine coordinates	X coordinate of point last drawn relative to RORIGIN. If text was drawn last, this point is the upper left corner of the character cell	All graphics instructions. Reset by RORIGIN.
RWHEREY	integer fine coordinates	Y coordinate of point last drawn relative to RORIGIN. If text was drawn last, this point is the upper left corner of the text cell.	All graphics instructions. Reset by RORIGIN.
SIZEX	integer	X size last specified or size when SIZE has only one argument.	SIZE instruction
SIZEY	integer	Y size last specified. A SIZE instruction with one argument changes this variable to the default y size associated with the argument.	SIZE instruction
WHERE	integer row and column coordinates	Row and column address of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell. After line graphics are displayed, the variable contains the address of the upper left corner of the text cell in which the last dot was drawn.	All graphics instructions. Reset to 0 by ERASE with no arguments.

Table C–1: Graphics and Graphing System Variables (Cont.)

Name	Data Type	Description	Modified By
WHEREX	integer fine coordinates	X coordinate of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell.	All graphics instructions. Reset to 0 by ERASE with no arguments.
WHEREY	integer fine coordinates	Y coordinate of last point displayed. After text is displayed, this point is one point to the right of the upper right corner of the character cell.	All graphics instructions. Reset to 0 by ERASE with no arguments.

Table C–2: Response-Related System Variables

Name	Data Type	Description	Cause of Update
ANSCNT	integer	Sequential number of RIGHT, RIGHTV, WRONG, or WRONGV matched by most recent response.	Cleared by QUERY. Set by first match and not reset until QUERY is encountered again. Set to 0 if no match.
ERRORV	integer	Value that identifies why the student failed to match the author's specified answer. Updated only if the matching process fails. See Chapter 9 for possible values.	Set by each RIGHT, WRONG, RIGHTV, or WRONGV instruction.
LATENCY	integer seconds	Time from QUERY or INPUT until delimiter is received.	Updated after every response and available until prompt is displayed again.

Table C–2: Response-Related System Variables (Cont.)

Name	Data Type	Description	Cause of Update
LENGTH	integer	Length in characters of most recent response.	Updated at end of each response.
PROMPT	string	Current prompt string. Contains > by default.	PROMPT instruction.
QELAPSED	integer seconds	Time that query is available. 0 indicates no time limit. QELAPSED counts down as time elapses. When it reaches 0, the variable TIMEOUT is set.	Set to 0 at beginning of lesson. Changed only if new value assigned.
QLENGTH	integer	Number of characters allowed in response. If this number is reached, response is terminated and judging begins.	Set to 0 (no limit) at beginning of lesson. Reset only if new value assigned.
RESPONSE	string maximum 500 chars	Student's exact response.	New value assigned at QUERY or INPUT. New response is terminated by DELIMIT character or value of QLENGTH. Characters entered later are read by next INPUT or QUERY if typeahead is allowed.
RESPONSEV	real integer Boolean	Student's response when the response is calculated as an expression. Data type is determined by the calculation required.	Evaluated when RIGHTV or WRONGV is executed.

Table C–2: Response-Related System Variables (Cont.)

Name	Data Type	Description	Cause of Update
TIMEOUT	Boolean	Used with QELAPSED to determine if time allowed for response has occurred. 0 = no timeout 1 = timeout	When QELAPSED is not 0, TIMEOUT is set to 0 at time of QUERY; to 1 when QELAPSED counts down to 0.
WORDS	integer	Number of words in response. Words are terminated by a space, a [illegible], or any punctuation mark except apostrophe, hyphen, dollar sign, or underscore. Other punctuation marks are ignored, and are not part of a word.	Updated at end of response.

Table C–3: Scoring System Variables

Name	Data Type	Description	Cause of Update
GOAL restart	integer 1 to 100	Number of current goal.	GOAL instruction
NNO restart	integer	Number of responses judged incorrect since the beginning of the lesson.	Incremented when unit containing QUERY returns with answer judged NO or by SCORE UPDATE.
NOK restart	integer	Number of responses judged correct since the beginning of the lesson.	Incremented by a right answer. Updated when unit containing QUERY returns, or by SCORE UPDATE.

Table C–3: Scoring System Variables (Cont.)

Name	Data Type	Description	Cause of Update
NOKFIRST restart	integer	Number of first responses judged correct since the beginning of the lesson.	Incremented by a correct response on the first attempt. Updated when unit containing QUERY returns, or by SCORE UPDATE.
NUMTRIES	integer	Number of times the student has responded to the current QUERY.	Incremented at each response. Cleared at beginning of a unit.
QUERIES restart	integer	Number of QUERY instructions since the beginning of the lesson. QUERIES is not incremented when prompt is redisplayed after a wrong response. It is incremented if any instruction (REDO, LOOP, FOR, BRANCH) returns control to a point preceding the QUERY.	Updated by each new QUERY instruction.
SATISFIED	integer	Assigned the value 1 if RESPONSE is judged correct, 2 if RESPONSE is judged incorrect, and 0 if the unit does not contain a QUERY block.	Updated after each RIGHT, WRONG, RIGHTV, or WRONGV instruction. Cannot be passed to another unit.
SCORE	real restart	Student's total score. Default increment is 1. Increment is modified by argument to WEIGHT instruction.	Updated when unit containing QUERY returns, or by SCORE UPDATE.

Table C–3: Scoring System Variables (Cont.)

Name	Data Type	Description	Cause of Update
SCORES	real restart	Array of SCORES. Indexed by goal number and containing score for each goal. The index is an integer between 1 and 100. If any other index is used, the value –999 is returned.	Updated when unit containing QUERY returns, or by SCORE UPDATE.

Note: Each unit containing a query is scored once. If instructions within the unit cause the query to be repeated, only the last response is scored.

Table C–4: Timing and Miscellaneous System Variables

Name	Data Type	Description	Cause of Update
ACCNAME	string 12 chars uppercase, left-justified, blank-filled	User's C.A.S name.	Read from C.A.S. files. When lessons are executed from VAX/VMS command level, this variable is null.
COURSE	string 9 chars, uppercase, left-justified, blank-filled	User's C.A.S group.	Read from C.A.S. files. When lessons are executed from VAX/VMS command level, this variable is null.
CUNIT	string maximum 32 characters, uppercase	Name of unit currently being executed.	Set when execution begins.

Table C–4: Timing and Miscellaneous System Variables (Cont.)

Name	Data Type	Description	Cause of Update
DATE	string DD-MON-YEAR	Current date.	VAX/VMS system service.
DTSTATUS	integer	Value indicating the status of the last DECtalk operation. See Appendix A in the VAX DAL Reference Manual for a table of possible DTSTATUS values.	Set after each DECtalk operation.
ECHO	Boolean	Controls whether user's responses are echoed to screen. 1 = echo 0 = no echo	Default is 1. Reset only if lesson assigns new value. Use for maintenance of current DAL code ONLY. Use SET ECHO instruction in new lessons.
ELAPSED	real seconds	Amount of time since beginning of current lesson.	Derived from the time stored at the start of the lesson and the current system time.
INTERRUPT	string maximum 32 characters, uppercase	Name of unit being executed at the time WHEN occurred. Value is null except during execution of unit called by interrupt.	WHEN instruction

Table C–4: Timing and Miscellaneous System Variables (Cont.)

Name	Data Type	Description	Cause of Update
IORESULT	integer	A numeric code that reports the status of the most recent I/O request. 1 = normal return 2 = invalid channel number 3 = insufficient virtual memory 4 = channel not open 5 = file is read only 6 = file is sequential only 7 = file is write only 8 = new file was created with OPEN instruction 9 = tray not opened (SLIDE expects previously opened tray file) 10 = RMS Error, check RMSSTATUS for error value 11 = file wrong type for operation 12 = file not indexed, index operation requested 13 = problem with key on FIND or GET 14 = restart file not found	OPEN instruction CLOSE instruction GET instruction PUT instruction FIND instruction UPDATE instruction DELETE instruction TRAY instruction SLIDE instruction CHARSET instruction MLOAD instruction FIND function EOF function REWIND function PICFILE function

Table C–4: Timing and Miscellaneous System Variables (Cont.)

Name	Data Type	Description	Cause of Update
		15 – restart variable missing 16 – permanent variable missing a 5- or 6-digit VAX/VMS error code is reported by any VMS error. (See note at end of table.)	
KEYPRESSED	integer	A numeric code that identifies the last key pressed by the user. Characters with ASCII values between 1 and 126 register ASCII values in this variable. Characters with MCS values between 128 and 255 register MCS values.	Set each time a key is pressed at the keyboard.
NAME	string, 20 chars, uppercase, left-justified, blank-filled	User's real name.	Read from CAS files. If lesson is executed from VAX/VMS command level, this variable is null.
ONCHANNEL	integer	Channel involved in a file-related error.	Set by each file-related error. Maintained by DAL.

Table C–4: Timing and Miscellaneous System Variables (Cont.)

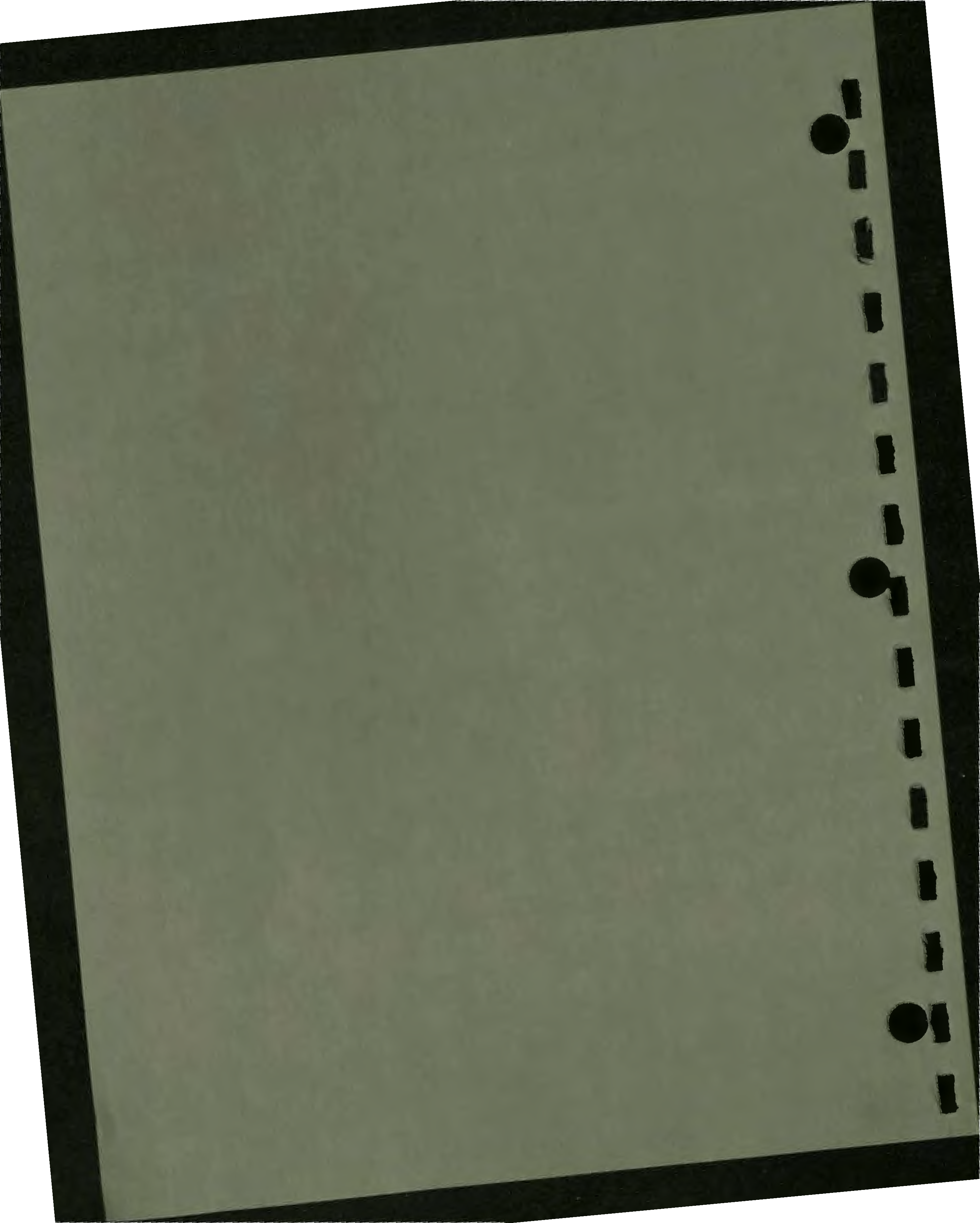
Name	Data Type	Description	Cause of Update
ONCODE	integer	Condition value; useful in the ANYCONDITION handler. Contains VMS code identifying a file I/O error (see VAX/VMS documentation for values)	Maintained by DAL.
ONKEY	integer	The key of a record involved in an indexed file error. Tied into indexed file support.	Set by an indexed-file-related error. Maintained by DAL.
RMSSTATUS	string	RMS status code of the last RMS operation.	An RMS operation.
TERMSTAT	integer	A read-only array containing complete data about the status of the terminal. See Appendix A in the VAX DAL Reference Manual for a list of TERMSTAT array entries.	Maintained by various DAL commands.
TIME	string HH:MM:SS.HH	Current time according to machine clock.	VAX/VMS system service.
TSA	array	Information about the secondary device attributes of the terminal (firmware revision level, options installed, and so on).	Obtained from VAX/VMS. See your terminal's Programmers Reference Guide for more information.

Table C–4: Timing and Miscellaneous System Variables (Cont.)

Name	Data Type	Description	Cause of Update
USERTYPE	string	Letter indicating type of user: A – author S – student G – group instructor	Read from C.A.S. files. When lessons are executed from VAX/VMS command level, this variable is null.

Note: Attempts to access files can return status codes from a number of VAX/VMS system services. For more information about error messages, refer to the *System Messages and Recovery Procedures Manual* in the VAX/VMS documentation set. The explanation of the SET MESSAGE command in the *VAX/VMS DCL Dictionary* explains how to convert the number returned in IORESULT to a system message. When the number in IORESULT is displayed in a DAL lesson, it is a decimal number.

D
Operators



D

Operators

Table D–1 defines the operators in DAL. The order of precedence is listed below the table.

Table D–1: Operators

Operator	Meaning	Data Type Operated On	Data Type Of Result
+	addition	integer, real	same
+	concatenation	string	string
–	negative	integer, real	same
–	subtraction	integer, real	same
*	multiplication	integer, real	same
/	division	integer, real	same
*	exponent	integer, real	integer, real
=	equal	integer, real, Boolean, string	Boolean
<>	not equal	integer, real, Boolean, string	Boolean

Table D–1: Operators (Cont.)

Operator	Meaning	Data Type Operated On	Data Type Of Result
<	less than	integer, real, Boolean	Boolean
<=	less than or equal to	integer, real Boolean	Boolean
>	greater than	integer, real Boolean	Boolean
>=	greater than or equal to	integer, real Boolean.	Boolean
AND	logical AND	Boolean	Boolean
OR	logical OR	Boolean	Boolean
XOR	logical XOR	Boolean	Boolean
IMP	logical IMP	Boolean	Boolean
NOT	logical inverse	Boolean	Boolean
MOD	Modulo	integer	integer

The evaluation of expressions follows algebraic conventions as far as possible: that is, expressions are evaluated from left to right following the order of precedence of the operators. The order of precedence can be overridden with pairs of parentheses.

The order of precedence is listed below. The highest precedence is 1.

1 Function evaluation (that is, returning the value of a function such as COS(x))
  Unary plus and minus (that is, specifying positive or negative numbers)
  Logical NOT

2 Exponentiation
  The logical operators =, <>, <, >, <=, and >=

3 The logical operators IMP, AND, XOR, and MOD
  Multiplication and division

4 The logical operator OR
  Addition and subtraction

E

Syntax Symbols

E

Syntax Symbols

Table E–1: Syntax Symbols

Symbol	Name	Description
.	period	1. in first character position of indented lines in QUERY blocks, IF structures, LOOP structures, FOR structures, and TEST structures; the period is followed by one TAB or one SP and is called "dot indentation" throughout this manual. 2. as a decimal point in real numbers
,	comma	general purpose separator; typical uses are between variable names used as arguments to the DEFINE instruction, between the x-coordinate and the y-coordinate that define one screen position, and between variable names used as arguments to system functions
;	semicolon	1. between two or more sets of x,y position arguments 2. in the first character position of a line as a symbol defining a comment line
:	colon	1. between position and other arguments in graphics instructions 2. between variable name and data type with DEFINE

Table E–1: Syntax Symbols (Cont.)

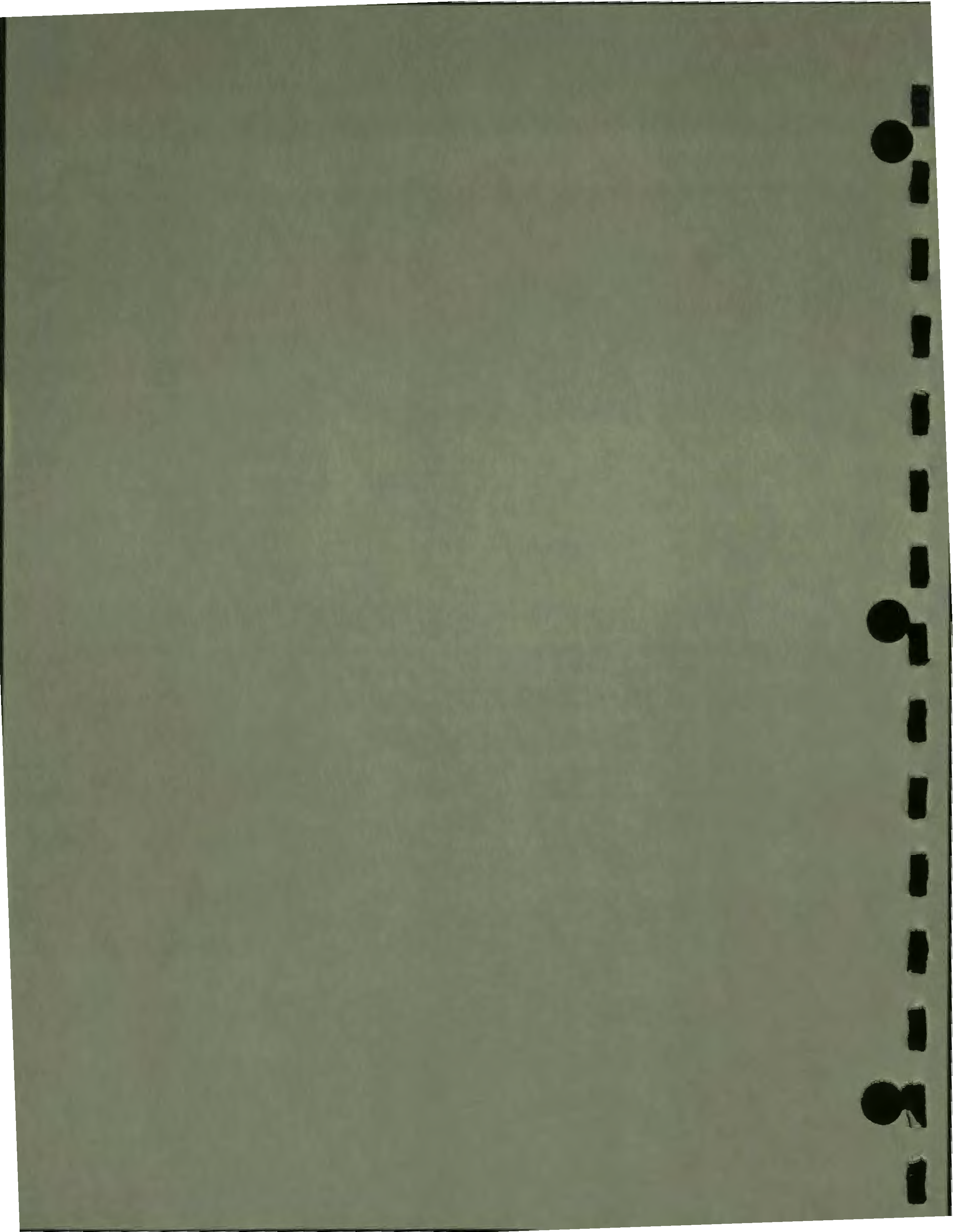
<table>
<tr><th>Symbol</th><th>Name</th><th>Description</th></tr>
<tr><td>!</td><td>exclamation point</td><td>in the first character position of a line as a symbol defining a comment line</td></tr>
<tr><td>=</td><td>equal sign</td><td>1. between constant name and value with DEFINE</td></tr>
<tr><td>:=</td><td>colon & equal</td><td>1. between variable name and value with ASSIGN
2. following counter with FOR</td></tr>
<tr><td>|</td><td>vertical line</td><td>between right answers or wrong answers used as arguments to RIGHT and WRONG</td></tr>
<tr><td>( )</td><td>parentheses</td><td>1. in expressions with the usual algebraic grouping
2. around the arguments to functions</td></tr>
<tr><td>[ ]</td><td>square brackets</td><td>1. around integer subscripts for arrays
2. around string subscripts for tables</td></tr>
<tr><td><< >></td><td>double angle brackets</td><td>1. around format specification and variable name with WRITE
2. around the word to be checked for synonyms with RIGHT and WRONG
3. in noise word position with RIGHT and WRONG</td></tr>
<tr><td>" "</td><td>double quotes</td><td>around string constants except as arguments to WRITE</td></tr>
</table>

Table E–1: Syntax Symbols (Cont.)

Symbol	Name	Description
%	percent sign	1. preceding a number, a percent sign and letter specifies the radix of the number: %B = binary %O = octal %D = decimal (default) %H = hexadecimal 2. in the first character position, a percent sign signifies a preprocessor directive
$	dollar sign	the first character in a label
$$	double dollar sign	Indicating beginning of comment on line with instruction

F

Sample Lessons



F

Sample Lessons

MULTIPLY

```
LESSON   multiply
DEFINE   go_on:BOOLEAN                    $$ True until student quits.
ASSIGN   go_on := TRUE
DEFINE   done_once:BOOLEAN                $$ Used in unit practice.
DEFINE   x,y,z:INTEGER                    $$ Used for all multiplication.

DO       intro1                           $$ Display title page
SCORE    FALSE                            $$ Turn off scoring.

; Set up loop. Lesson returns here after each practice or
; review. Loop is broken and lesson ends when student chooses to quit.
LOOP     go_on                            $$ Begin loop.
.        DO       menu                    $$ Display instructions and choices.
.        QUERY
.        RIGHT    practice
.        .        DO       practice
.        RIGHT    quit
.        .        DO       quit
.        .        ASSIGN   go_on := FALSE
.        RIGHT    REVIEW
.        .        DO       review
.        WRONG                            $$ Anything else is wrong.
.        .        SIZE     1              $$ Display error message
.        .        WRITE    YOU MUST TYPE ONE OF THE WORDS ABOVE
.        .        SIZE     2              $$ and return to beginning
.        ENDQ                             $$ of QUERY block.
ENDLOOP                                   $$ end of loop
```

```
; End of lesson level.

UNIT    intro1
ERASE
FCOLOR  RED
BOX     0,0;767,479:-12          $$ Draw red box at limits of screen.
FCOLOR  YELLOW
BOX     12,12;755,467:-12        $$ Draw yellow box inside red.
FCOLOR  RED
BOX     24,24;743,455:-12        $$ Draw red inside yellow
SIZE    4,8
AT      1115
WRITE   MULTIPLICATION
DO      return                   $$ Execute unit return

; End of unit intro1. Unit return returns to this point. Because
; this is end of unit intro1, unit intro1 returns to instruction
; following DO intro1 at lesson level

UNIT    menu
FCOLOR  RED
ERASE
BOX     0,0;0.999,0.999          $$ For manual illustrations.
AT      210
SIZE    2
WRITE   This lesson lets you review your
        multiplication tables or practice
        some multiplication problems.
AT      820
WRITE   Do you want to
AT      1125
WRITE   PRACTICE
SIZE    1
AT      1332
WRITE   OR
SIZE    2
AT      1427
WRITE   REVIEW
SIZE    1
AT      1632
WRITE   OR
SIZE    2
AT      1729
WRITE   QUIT
```

```
UNIT     review              $$ Select to review one or all tables.
ERASE
BOX      0,0;0.999,0.999            $$ For manual illustrations.
SIZE     2
AT       510
WRITE    Type ALL to review all
         the multiplication tables.

         Type a number to review
         that multiplication table.
QUERY
RIGHT    all
.        ERASE
.        BOX      0,0;0.999,0.999          $$ For manual illustrations.
.        DO       revall                   $$ Review all.
RIGHT    1|2|3|4|5|6|7|8|9|10|11|12
.        ASSIGN   x := NUMBER(RESPONSE)
.        ERASE
.        BOX      0,0;0.999,0.999          $$ For manual illustrations.
.        DO revone                         $$ Review chosen one.
WRONG
.        SIZE     1
.        WRITE    You must type ALL or a number from 1 to 12.
                  Try again.
.        SIZE     2
ENDQ

; The unit revone displays one times table. It is called 12 times
; by unit revall or once by unit review.
UNIT     revone              $$ The value of x when revone is called is
AT       025                 $$ set by unit review or unit revall.
FCOLOR   BLUE
SIZE     2
FOR      y := 0,12                  $$ For 13 times, changing the value
.        ASSIGN   z := x*y          $$ of y each time, calculate answer
.        WRITC    <<T,x,2,0>> x <<T,y,2,0>> = <<T,z,3,0>>
ENDFOR                              $$ and display it.
DO       return                     $$ Execute unit return.

UNIT     return                     $$ Display press return and
SIZE     1                          $$ wait at pause.
AT       2060
WRITE    PRESS
MODE     INVERSE
WRITE    RETURN
PAUSE
ERASE
MODE     NORMAL
BOX      0,0;0.999,0.999
```

```
UNIT     quit                              $$ Display BYE.
ERASE
BOX      0,0;0.999,0.999                   $$ For manual illustrations.
AT       1020
SIZE     10
FCOLOR   RED
WRITE    BYE
DO       return

UNIT     revall               $$ Reviews all times tables.
FOR      x:=1,12              $$ For 12 times, increment x and
.        DO        revone     $$ display one table.
ENDFOR

UNIT     PRACTICE             $$ Displays 25 problems and scores them.
DEFINE   old_x,old_y:INTEGER
ASSIGN   old_x := 3           $$ Used to see if current x and y are
ASSIGN   old_y := 4           $$ same as x and y for last problem.
ERASE
BOX      0,0;0.999,0.999                   $$ For manual illustrations.
FCOLOR   RED
SIZE     2

; The IF block is executed only the first time the student chooses
; to practice. The variable done_once is defined at lesson level and
; is assigned the value TRUE in this unit.
IF       done_once = FALSE                 $$ If this is the first time
.        do        instruc                 $$ display instructions
ENDIF

SCORE    TRUE      $$ Begin scoring.

DEFINE   c:INTEGER            $$ Counter for problems.
SEED
FOR      c := 1,25                         $$ Do 25 problems.
.        ASSIGN    X:=RANDOMU(0,13)             $$ Assign values to x and y.
.        LOOP      x = old_x
.        .         ASSIGN X := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN    old_x := x
.        ASSIGN    Y:=(RANDOMU(0,13))
.        LOOP      y = old_y
.        .         ASSIGN    y := RANDOMU(0,13)
.        ENDLOOP
.        ASSIGN    old_y := y
.        DO        prac1                   $$ Display problem.
.        DO        return
ENDFOR
```

```
SCORE    FALSE                $$ Stop scoring.
DO       shoscore             $$ Display score for practice.
GOAL     GOAL+1               $$ Increment goal for next time.

UNIT     shoscore
FCOLOR   RED
SIZE     2
AT       505
TEST     SCORES(GOAL)         $$ Score for set of 25 problems.
VALUE    25                   $$ All right.
.        WRITE     VERY GOOD
                   YOU GOT THEM ALL RIGHT.
VALUE    20..24
.        WRITE     GOOD
                   YOUR SCORE IS <<S,INT(SCORES(GOAL))>>.
VALUE    0..19
.        WRITE     YOU SHOULD REVIEW BEFORE
                   YOU PRACTICE AGAIN.
                   YOUR SCORE IS <<S,INT(SCORES(GOAL))>>.
ENDTEST
AT       1305

; If student has practiced before, tell him how this time compares
; to the last time.
; Outer IF tests for more than one time. Next IF tests for different
; scores. If scores are different, third IF tests for which is
; better, and displays appropriate message. If scores are same,
; word TOO is displayed following score for last time.

IF       GOAL > 1                       $$ Practiced more than once.
.        WRITC     LAST TIME YOUR SCORE WAS <<s,int(scores(goal-1))>>
.        IF        SCORES(GOAL) <> SCORES(GOAL-1)
.        .         IF SCORES(GOAL) > SCORES(GOAL-1)
.        .         .         WRITC     YOU'RE DOING BETTER
.        .         ELSE
.        .         .         WRITC     YOU'VE SHOWN YOU CAN DO BETTER
                                       TRY IT AGAIN.
.        .         ENDIF
.        ELSE
.        .         WRITE     TOO.
.        ENDIF
ENDIF
DO       return                         $$ Execute unit return
```

```
UNIT      PRAC1                       $$ Unit prac1 displays and judges one problem.
ASSIGN    Z:=X*Y                      $$ calculate right answer for these
PROMPT    "= "
FCOLOR    BLUE                        $$ two numbers
SIZE      3
AT        820                         $$ display current problem
WRITE     <<S,x>> x <<s,y>>
QUERY
RIGHTV    Z                           $$ the current value of Z is right
.         AT       1420
.         SIZE     2
.         WRITE    You're right,
.         AT       1720
.         SIZE     3
.         WRITE    <<S,x>> x <<S,y>> = <<S,z>>
WRONG                                 $$ anything else is a wrong answer
.         AT       1420
.         SIZE     2
.         FCOLOR   RED
.         WRITE    NO, the right answer is
.         SIZE     3
.         AT       1720               $$ display the right answer
.         WRITE    <<S,x>> x <<S,y>> = <<S,z>>
.         JUDGE    STOP               $$ stop judging, do not repeat query
ENDQ
PROMPT    ">"

UNIT      instruc
ASSIGN    X:=3     $$ assign values for practice problem.
ASSIGN    Y:=4
AT        503
WRITE     You will see 25 multiplication problems.
          Type the answer and press RETURN.
          You will be told if
          you got the answer right.

          Press RETURN again for the next problem.
          Press RETURN now to see a sample.
```

```
DO        return

LOOP      done_once = FALSE           $$ Display problem and loop
.         DO        prac1             $$ until student gets it right
.         SIZE      2
.         TEST      RESPONSEV         $$ It was right.
.         VALUE     12
.         .         AT        310
.         .         WRITE     GOOD -- you seem to understand.
                    Now the real problems start.
.         .         DO        return
.         .         ASSIGN    done_once:=TRUE
;                                     $$ Assign new value
.         OTHER                       $$ to control variable.
.         .         AT        310
.         .         WRITE     Try again, now that you know
                              the right answer
.         .         DO        return
.         ENDTEST
ENDLOOP

ENDLESSON                     $$ end of all instructions in lesson file
```

ICECREAM

The lesson Icecream requires the auxiliary file GLOB.FNT for the character set used for the chocolate chips and the drip from the ice cream. The file GLOB.FNT must be in the same directory as the executable image of Icecream.

```
LESSON    ICECREAM
DEFINE    CHIPS:INTEGER
DEFINE    CONTINUE:BOOLEAN
ASSIGN    CHIPS:=0
SYN       +"DARK","LICORICE","CHOCOLATE","COFFEE"
SYN       +"BLUE","BOYSENBERRY"
SYN       +"WHITE","VANILLA"
SYN       +"GREEN","LIME","PISTACHIO","MINT"
SYN       +"RED","STRAWBERRY","CHERRY"
SYN       +"YELLOW","LEMON","BANANA"
SYN       +"MAGENTA","RASPBERRY","PEPPERMINT"
SYN       +"CYAN","BLUEBERRY"
SPECS     ANYORDER
SPECS     EXTRA
ERASE
FCOLOR    BLUE
SIZE      4
AT        510
```

```
WRITE     ICE CREAM STORE
CHARSET   "GLOB.FNT"
CHARSET   "STANDARD"

ASSIGN    CONTINUE := TRUE
LOOP      CONTINUE
.         DO        ASK
.         DO        CONE
.         DO        REP
ENDLOOP
;end of lesson level

UNIT      ASK
SIZE      2
FCOLOR    YELLOW
AT        1515
WRITE     What flavor do you want?
QUERY
RIGHT     CHIP
.         ASSIGN    CHIPS:=1
.         JUDGE CONTINUE
RIGHT     <<GREEN>>
.         FCOLOR  GREEN
RIGHT     <<WHITE>>
.         FCOLOR  WHITE
RIGHT     <<RED>>
.         FCOLOR  RED
RIGHT     <<YELLOW>>
.         FCOLOR  YELLOW
RIGHT     <<BLUE>>
.         FCOLOR  BLUE
RIGHT     <<MAGENTA>>
.         FCOLOR  MAGENTA
RIGHT     <<CYAN>>
.         FCOLOR  CYAN
RIGHT     <<DARK>>
.         FCOLOR  DARK
WRONG
.         FCOLOR  WHITE
.         ERASE     1310;2080
.         AT        1310
.         WRITE     SORRY, we're out of <<s,response>>
                    Order another flavor
ENDQ
SIZE      1
```

```
UNIT      CONE
DEFINE    J,I,old_fcolor:INTEGER
ERASE
; Change colors for chocolate chip
IF        CHIPS=1 AND FCOLOR = DARK
.         BCOLOR  DARK
.         FCOLOR  WHITE
ENDIF
; Change background if fcolor dark
IF        FCOLOR = DARK AND CHIPS=0
.         BCOLOR  WHITE
ENDIF
ASSIGN    old_fcolor := fcolor
SREF      NONE,120                              $$ Draw the icecream.

CIRCLE    360,120:100,0,180
CHARSET   "GLOB.FNT"
IF        CHIPS=1
.         MODE     COMPLEMENT
.         AT       290,30
.         WRITE     b  c
                    b  c     c
                    c    b  c
                   c       b       b
                    b      c
.         MODE     REPLACE
ENDIF
FCOLOR    YELLOW
LINE      460,120;360,440               $$ Draw the cone.
LINE      260,120
SREF
ASSIGN    CHIPS:=0
FOR       I:=0,70,10
.         IF       BCOLOR = WHITE               $$ Change foreground color
.         .        FCOLOR  white                $$ to background color for
.         ELSE                                  $$ licking the icecream.
.         .        FCOLOR  DARK
.         ENDIF
.         MODE     ERASE
.         FOR      J:=0,8
.         .        CIRCLE   360,250:230-I-J,45,135
.         ENDFOR
.         MODE     COMPLEMENT
.         RORIGIN  360,160
.         FCOLOR   old_fcolor           $$ Change foreground color again
.         FOR      J:=160,440,20        $$ for the drop.
.         .        RAT      -120,0
```

```
.       .       WRITE   a
                        d
.       .       RAT     -120,0
.       .       WRITE   a
                        d
.       .       RORIGIN 360,J
.       ENDFOR
ENDFOR
CHARSET "STANDARD"
PAUSE   ELAPSED,1
FCOLOR  BCOLOR
MODE    ERASE                   $$ Take bites out of cone.
SREF    NONE,120
CIRCLE  450,120:100

SREF    NONE,130
CIRCLE  300,130:60
PAUSE   ELAPSED,1
SREF    NONE,470
CIRCLE  360,440:60
PAUSE   ELAPSED,1
ERASE
SIZE    10
MODE    REPLACE
AT      20,240
FCOLOR  RED
WRITE   GULP!!
PAUSE   ELAPSED,1
ERASE
AT      20,140
FCOLOR  BLUE
WRITE   BURP??
PAUSE   ELAPSED,1

UNIT    REP
ERASE
SIZE    3
AT      315
WRITE   I scream, You scream
        We all scream for
SIZE    5
AT      1220
FCOLOR  RED
WRITE   ICE CREAM
SIZE    2
```

```
AT      2020
WRITE   Do you want more?
QUERY
RIGHT   YES | Y
.       BCOLOR  DARK
.       ERASE
.       AT      1320
.       SIZE    2
.       FCOLOR  YELLOW
.       WRITE   This time,
WRONG   NO | N
.       ASSIGN  CONTINUE:=FALSE
.       JUDGE STOP
ENDQ

ENDLESSON
```

MENU_DRIVER

Lesson Menu_Driver is a simple menu driver program that uses special function keys to place an arrow beside the menu topic to be chosen. Notice also that Menu_Driver consists of two VAX DAL modules that are compliled separately and then linked together. This requires the use of global and external variables.

```
LESSON  DM
!  This lesson consists of two separately compiled modules:
!                 DM and UTILITY.
!  These should be compiled separately and then linked together.
!
!  This lesson demonstrates the use of the special function keys
!  to manage a menu of items on the screen.
!
DEFINE  whichone : integer, global        $$ which item was picked
DEFINE  menuitems[6] : string             $$ array contains menu topics

; Menu topics.
ASSIGN  menuitems[1] := "Building the Foundation"
ASSIGN  menuitems[2] := "Framing the Structure"
ASSIGN  menuitems[3] := "Roofing the Structure"
ASSIGN  menuitems[4] := "Wiring the Structure"
ASSIGN  menuitems[5] := "Plumbing the Structure"
ASSIGN  menuitems[6] := "Finishing the Interior"
```

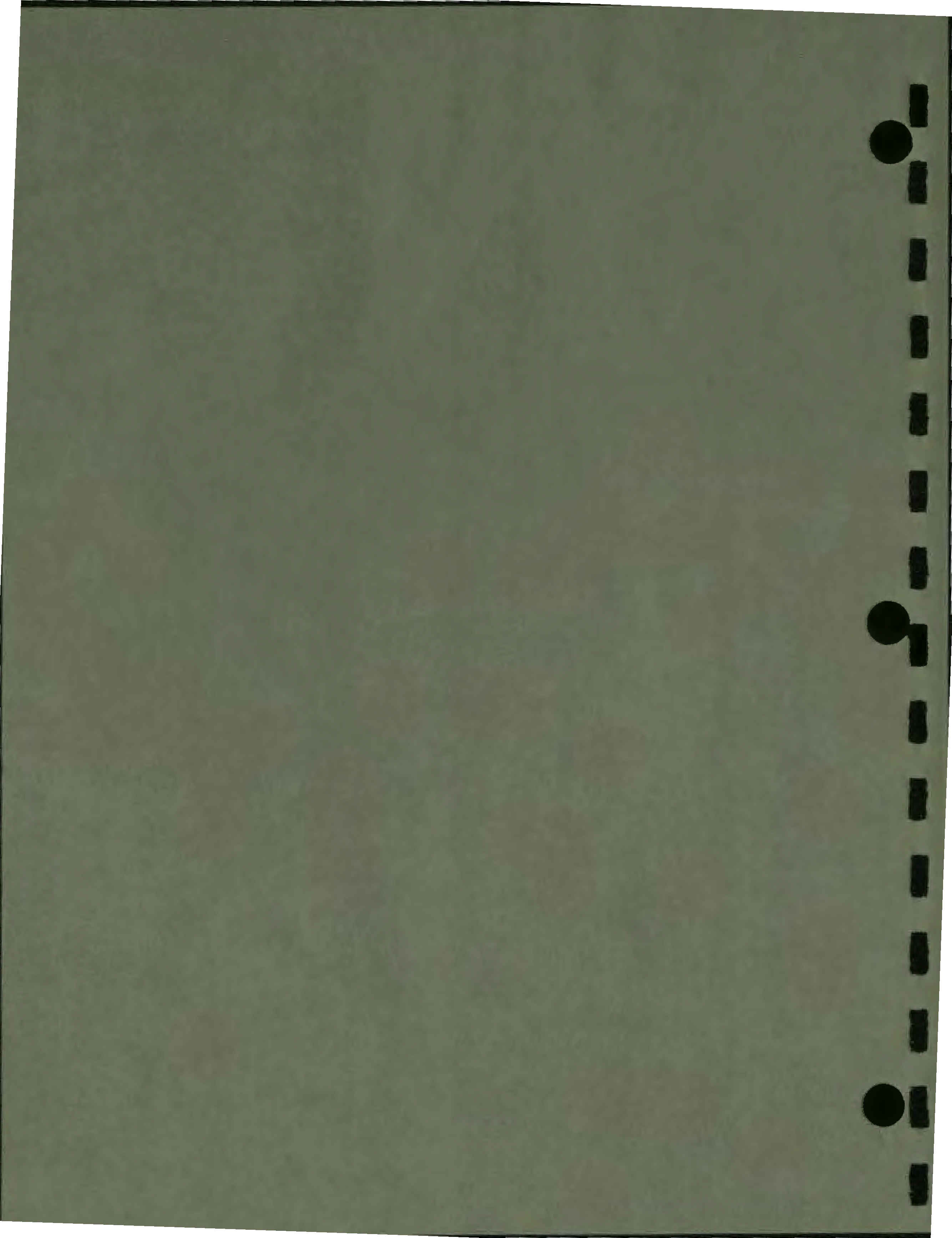
```
SET     FKEY, TERMINATE              $$ will be using special function keys
SET     KEYPAD, NUMERIC              $$ set the keypad to numeric mode
SET     TYPEAHEAD, OFF               $$ typeahead is not allowed
ERASE
AT      300
SIZE    3
FCOLOR  yellow
WRITE   CONSTRUCTION DEMONSTRATIONS
AT      610
SIZE    1
FCOLOR  cyan
WRITE   This is a collection of CBI lessons about various
        construction topics. You may select a topic and
        and then select a demonstration lesson.

        These lessons are for demonstration purposes only.
DO      pressret

DO      mainmenu

AT      1810
WRITE   You selected: <<s,whichone>>–<<s,menuitems[whichone]>>
PAUSE

UNIT    mainmenu
ERASE
AT      300
SIZE    3
FCOLOR  yellow
WRITE   Construction Topics
FCOLOR  cyan
AT      615
SIZE    1
WRITE   Use the UP and DOWN arrow keys to select a topic,
        and then press the SELECT key to select the topic.

DO      getitem(menuitems,10,35,30)           $$ call driver routine
RETURN

ENDLESSON
```

```
MODULE  utility
!
!  A separately compiled module which is part of the DM lesson.
!
!  Contains the following units:
!          GETITEM
!          PRESSRET
!
!  Global/External variables:
!
DEFINE    whichone : integer, external
!
UNIT      getitem (array,row,col,pcol)
!
!   This unit is a simple menu driver.
!
!   It requires four parameters:
!
!    array – name of the array containing the menu items
!      row – the row the menu items are to begin on
!      col – the column the menu items are to begin in
!     pcol – the column to display the prompt in
!                 ( the prompt is "->" )
!
DEFINE    array[?] : string
DEFINE    row,col,pcol : integer
DEFINE    z,num : integer
DEFINE    gotit: boolean              $$ TRUE if item has been selected

SET       ECHO, OFF                   $$ don't echo input
ASSIGN    gotit := false
ASSIGN    num := dimax(array,1)       $$ NUM is the number of items to display
!
! First write out the menu
!
```

```
FCOLOR  cyan
FOR     z := 1,num
.       AT      (row+z-1)*100+col
.       WRITE   <<s,array[z]>>
ENDFOR
!
! Now prompt and get an item
!
FCOLOR  yellow
AT      (row-1)*100+pcol
WRITE   ->
PROMPT  ""
LOOP    not gotit
.       INPUT   *
.       TEST    response
.       VALUE   "[E04_KEY]"         $$ SELECT key pressed
.       .       IF (int(where/100) >= row)          $$ check for valid choice
                AND (int(where/100) <= (row+num-1))
.       .       .       assign gotit := TRUE
.       .       .       assign whichone := int(where/100)-9
.       .       ENDIF
.       VALUE   "[DNA_KEY]"         $$ DOWN arrow pressed
.       .       TEST    int(where/100)
.       .       VALUE   (row-1)..(row+num-2)
.       .       .       AT      int(where/100)*100+pcol
.       .       .       MODE    erase
.       .       .       WRITE   ->
.       .       .       MODE    replace
.       .       .       AT      (int(where/100)+1)*100+pcol
.       .       .       WRITE   ->
.       .       OTHER   $$ pointer was at bottom of menu; move to top item
.       .       .       AT      int(where/100)*100+pcol
.       .       .       MODE    erase
.       .       .       WRITE   ->
.       .       .       MODE    replace
.       .       .       AT      row*100+pcol
.       .       .       WRITE   ->
.       .       ENDTEST
.       VALUE   "[UPA_KEY]"         $$ UP arrow pressed
.       .       TEST    int(where/100)
.       .       VALUE   (row+1)..(row+num-1)          $$ move pointer up one item
```

```
.       .       .       AT      int(where/100)*100+pcol
.       .       .       MODE    erase
.       .       .       WRITE   ->
.       .       .       MODE    replace
.       .       .       AT      (int(where/100)-1)*100+pcol
.       .       .       WRITE   ->
.       .       OTHER   $$ pointer was at top of menu; move to bottom item
.       .       .       AT      int(where/100)*100+pcol
.       .       .       MODE    erase
.       .       .       WRITE   ->
.       .       .       MODE    replace
.       .       .       AT      (row+num-1)*100+pcol
.       .       .       WRITE   ->
.       .       ENDTEST
.       ENDTEST
ENDLOOP
SET     ECHO, ON        $$ turn echo back on
RETURN

;
;       This unit causes a message saying
;           Press RETURN to be printed
;
UNIT    pressret
DEFINE  oldfcolor : integer
DEFINE  oldsizex,oldsizey : integer
!
! Save old attributes.
!
ASSIGN  oldfcolor := fcolor
ASSIGN  oldsizex := sizex
ASSIGN  oldsizey := sizey
SIZE    1
FCOLOR  yellow
AT      2360
MODE    inverse
WRITE   Press RETURN to continue
PAUSE
MODE    normal
!
! Restore old attributes.
!
FCOLOR  oldfcolor
SIZE    oldsizex,oldsizey
RETURN

ENDMODULE
```

Glossary



Glossary

answer: a number or text string, expressed as a constant or a variable, that is an argument to one of the instructions RIGHT, RIGHTV, WRONG, or WRONGV. The author specifies possible answers, either right or wrong. (See RESPONSE.)

anticipated response: a student's response that matches one of the specified answers. The author has anticipated that students will enter this response, and has specified it as a right answer or as a wrong answer.

current record: a record in an external file at which control is positioned. The current record is the record that is used in a file I/O operation.

current unit: a unit that is currently executing in a lesson.

fine coordinates: screen address coordinates that define a location on the screen as one of 768 horizontal addresses (x-coordinates) and one of 480 vertical addresses (y-coordinates).

graphics: pertaining to pictures or to drawing pictures. Graphics instructions in DAL draw lines, circles, dots, boxes, arrows, and curves that can be combined into pictures or used to emphasize text.

graphing: pertaining to plotting graphs. Graphing instructions define the screen as a graph, plot points on the graph, and draw graphics whose addresses are points on the graph.

keyword: a word with a specified meaning when it is used as an argument to a DAL instruction.

module: a DAL lesson source file that is compiled apart from the source file containing the lesson-level instructions.

normalized coordinates: screen address coordinates that specify a location on the screen as a proportion of the total horizontal distance (x-coordinate) and the total vertical distance (y-coordinate).

permanent variable: a variable that is stored when execution of a lesson ends. There is one set of permanent variables for each lesson.

response: anything the student enters from the keyboard. (See ANSWER.)

response-contingent: depending on the student's response. This term is generally used to describe an instruction or a sequence of instructions that is executed only when a student enters a response that matches one of the specified answers.

response judging: the process of determining whether a response matches a specified answer. Response judging also includes scoring, executing response-contingent instructions, and the control logic that determines whether the question is repeated so the student can enter another response.

restart variable: a variable that is stored when execution of a lesson is stopped by the STOP instruction. There is a separate set of restart variables for each student who executes a lesson.

row and column coordinates: screen address coordinates that specify a location on the screen as one of 24 rows and one of 80 columns.

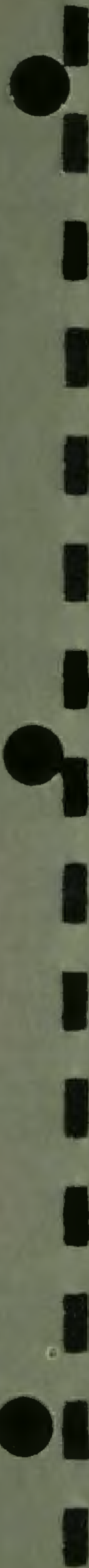
student: anyone who executes a lesson.

student variable: a variable whose name is available to students when they execute lessons.

system variable: a variable that is maintained by C.A.S. and is available to any lesson.

unanticipated response: any student response that does not match one of the specified answers. The author has not anticipated this response.

Index



Index

A

ABS function, B–1
ACCNAME variable, C–8
ALT function, B–1
ANSCNT variable, C–4
Answer, 1–1, 4–1, 4–7
  variable, 5–21, 8–8
ANTILOG function, B–1
ARCCOS function, B–1
ARCSIN function, B–1
ARCTAN function, B–1
ARCTAND function, B–1
Argument, 2–3
Arithmetic
  expressions, 2–8
  operators, 2–8
ASCII
  function, B–1
ASSIGN instruction, 5–6, 5–20, A–1
  example, 5–18
AT instruction, 3–3, A–1
  example, 3–9
Author, 2–1
AXES instruction, A–1

B

BACKUP instruction, 11–3, A–1
Baud rate, 10–37
BCOLOR instruction, 3–23, 12–9, A–1
BCOLOR variable, 10–32, C–1
Boolean
  expressions, 2–8, 5–7, 5–25, 5–33
  operators, 2–8
  variables, 5–6
BOX instruction, 10–2, 10–4, A–1
  example, 3–4
BRANCH instruction, 11–8, A–1

C

Calling chain, 11–2
CANCEL instruction, 11–20, A–1
Case
  in source code, 2–11
CCOLOR instruction, 12–8, A–1
CDUNIT instruction, 11–19, A–2
CHAR function, B–1
CHAR instruction, A–2
Character
  cell, 3–1
Character size, 3–18
CHARSET instruction, A–2
CHARSET variable, C–1
CHARSETLD function, B–2
CHECKERR
  blocks, 9–16
  checking ERRORV, 9–16
  instruction, 9–16, A–2
CIRCLE instruction, 10–2, 10–5, 10–6, A–2
  example, 3–7
CLIP instruction, A–2
CLOSE instruction, 13–8, A–2

Color, 3–23, 5–15, 10–37
Color capabilities
  terminal models, 7–4
Color constant names, 12–5
Color lessons on B & W terminals, 12–10
Color management system, 12–1
  example, 12–11
Color map, 12–3
  default contents, 12–4
Color palettes, 12–3
Color specifications, 12–2
  DAL-provided, 12–4
Color table, 12–5
Comment line, 2–11, 2–12
Compiler, 6–1, 6–7
  errors, 6–3
  switches, 6–7
COMPLEMENT mode, 10–23
Condition handler, 11–16
  canceling a, 11–20
  condition unit, 11–19
  declaring a, 11–16
  example, 11–21
  signaling a, 11–20
Conditional transfer of control, 11–8
Constants, 2–7
  data types, 2–7
  names, 2–10
  string, 8–14, 9–10
CONTROL instruction, A–2
Control logic
  conditional transfer of control, 11–8
  instructions, 11–1
  unconditional transfer of control, 11–3
Conventions, iii
CONVERT instruction, 8–12, A–2
COS function, B–2
COURSE variable, C–8
CUNIT variable, C–8
Current
  location, 3–11, 3–13, 10–34
  record, 13–9
  text size, 3–13, 5–11, 5–13
CURVE instruction, 10–2, 10–7, A–2

D

Data records, 13–2
Data type conversion, 5–19, 5–21
Data types, 2–4, 5–18
  BOOLEAN, 2–5
  constants, 2–7
  INTEGER, 2–4
  REAL, 2–4
  RECORD, 2–5
  STRING, 2–4
DATE variable, C–9
DECmate II and III terminals
  color capabilities, 7–5
  VAX DAL on, 7–2
DECTALK instructions, A–2
DEFINE instruction, 2–4, 2–7, 5–5, A–3
  example, 5–26
  fixed-length strings, 13–16
  record structures, 13–16
DEG function, B–2
DELETE function, B–2
DELETE instruction, 13–8, A–3
  indexed files, 13–24
Delimit character, 4–2, 9–7
DELIMIT instruction, A–3
DELTA instruction, A–3
DET function, B–2
DIMAX function, B–2
DIMS function, B–2
Display modes, 3–2, 3–24, 5–15, 10–20
Displaying
  variables, 3–15
DO * instruction, A–3
DO instruction, 2–2, 5–2, 5–6, 5–18, 5–21, A–3
  example, 5–27
  parameter passing, 14–3
Dot indentation, 2–12, 2–13, 5–7, 5–18
  response-judging block, 4–2
  WRITE instruction, 3–15
DOT instruction, 10–2, 10–10, A–3
  example, 3–7
DTSTATUS variable, C–9

E

ECHO variable, C–9
ELAPSED variable, C–9
ELSE instruction, 5–29, 5–30, A–3
ENDFOR instruction, A–3
ENDIF instruction, 5–29, A–3
ENDLESSON instruction, 5–40, A–3
ENDLOOP instruction, 5–7, A–3

%ENDMACRO instruction, 15–2
ENDMODULE instruction, A–3
ENDQ instruction, 4–2, A–4
ENDRECORD instruction, A–4
ENDTEST instruction, 5–34, A–4
EOF function, B–2
ERASE instruction, 3–14, 10–19, A–4
  example, 4–7
ERASE mode, 10–23
Errors
  compiler, 6–3
  run-time, 6–5
ERRORV variable, 9–6, 9–14, C–4
  checking the, 9–16
  error codes, 9–14
  expressions and, 9–18
  interpretation, 9–18
  multiword responses, 9–21
  setting of the, 9–15
EVAL function, B–2
Execution
  VAX/VMS run-time library routines, 14–6
  VAX/VMS system services, 14–7
EXP function, B–2
Expressions
  arithmetic, 2–8, 5–24
  Boolean, 2–8, 5–7, 5–25
  in responses, 8–9, 8–13
External variables, 2–5
  example, F–11

F

FCOLOR instruction, 3–23, 12–7, A–4
  example, 5–20
FCOLOR table_slot_number instruction, 12–7
FCOLOR variable, 10–32, C–1
  example, 10–34
Feedback, 4–5
File I/O, 13–1
  current record, 13–9
  data fields, 13–2
  data records, 13–2
  file structures, 13–3
  indexed files, 13–15
  random files, 13–11
  record structures, 13–16
  sequential files, 13–8
File input/output
  See also File I/O
File input/output instructions, 13–6
  CLOSE, 13–8
  DELETE, 13–8
  FIND, 13–7
  GET, 13–7
  OPEN, 13–6
  PUT, 13–7
  UPDATE, 13–7
FIND function, 13–9, B–3
FIND instruction, 13–7, A–4
  indexed files, 13–20
  random files, 13–13
Fine coordinates, 3–4, 3–10
  odd-even pairs, 3–5
FOR instruction, 5–25, A–4
  example, 5–23
FUNCT instruction, A–4
Function, 2–8
Function calls
  parameter passing with, 14–3

G

GAT instruction, A–4
GBOX instruction, A–4
GCIRCLE instruction, A–4
GCURVE instruction, A–4
GDOT instruction, A–4
GET instruction, 13–7, A–5
  indexed files, 13–17
  random files, 13–12
  sequential files, 13–9
GIGI (VK100) terminal
  color capabilities, 7–5
  VAX DAL on a, 7–2
GLINE instruction, A–5
Global variables, 2–5
  example, F–11
GOAL instruction, 5–24, A–5
GOAL variable, 5–24, 5–32, 9–2, C–6
GORIGIN instruction, A–5
GORIGINX variable, C–1
GORIGINY variable, C–1
GRAFSTAT variable, C–2
GRAPH instruction, A–5
Graphics, 1–2
  relative, 10–23
Graphics instructions, 10–1

Graphics system variables, 2–6, C–1
GVECTOR instruction, A–5
GWHEREX variable, C–2
GWHEREY variable, C–2

H

HBAR instruction, A–5
Hue/Lightness/Saturation (HLS) method, 12–3

I

IDEN function, B–3
IF instruction, 5–25, 5–29, 5–30, 5–31, 5–32, A–5
  example, 10–34
%INCLUDE instruction, 15–4
INCLUDE instruction, A–5
Indexed files
  access to records, 13–5
  file I/O, 13–15
  file structure, 13–5
  record structures, 13–16
INPUT instruction, 8–15, A–5
INSTRING function, B–3
Instruction, 2–2
INT function, 2–8, B–3
Integer variables, 5–6
Integers, 2–4
INTERRUPT variable, C–9
INV function, B–3
INVERSE mode
  example, 5–15
IORESULT variable, C–10
  values, 11–18
IS function, B–3
ITALICS instruction, 10–16, A–5
ITALICS variable, C–2

J

JUDGE instruction, 5–21, 5–38, 8–6, 8–8, 9–10, A–5
  example, 9–10
Judging student responses
  See Response

K

KEYPRESSED variable, C–11
Keywords, 2–3

L

LATENCY variable, C–4
LEN function, 13–14, B–3
LENGTH variable, C–5
Lesson, 1–1, 1–4, 2–1
  level, 2–1, 5–2
  name, 2–9
  transporatbility, 7–1
  use of function keys in, 7–8
Lesson flow, 11–1
  conditional transfer of control, 11–8
  unconditional transfer of control, 11–3
LESSON instruction, 5–2, 5–4, A–6
Lesson structure, 5–2
Lesson-level variables, 2–10, 5–5
LINE instruction, 10–2, 10–10, A–6
  example, 3–8
Line speed, 10–37
Linking, 6–1
  VAX DAL run-time library, 6–9
LIST/NOLIST compiler switch, 6–8
LN function, B–3
LOG function, B–3
LOG instruction, A–6
LOOP instruction, 5–7, 5–25, 5–38, A–6
  example, 5–23
LOWER function, B–3
Lowercase, 2–11
LSCALEY instruction, A–6

M

Machine error tolerance
  SPECS MACHTOL, 8–11
%MACRO instruction, 15–2
Macros, 15–1
  with parameters, 15–3
MAP instruction, 12–10, A–6
MARKUP instruction, 4–5, A–6
  example, 4–7
MARKY instruction, A–6

MATCH instruction, A–6
MGRAPH, ENDMGRAPH instruction, A–6
MLOAD instruction, A–6
MODE instruction, 3–24, 3–26, 5–15, 10–20, A–6
example, 3–12, 10–34
system constant, 3–26
Module, 2–2
names, 2–9
MODULE instruction, A–7
MPLOT instruction, A–7

N

NAME variable, C–11
Named constants, 2–7
example, 10–19
Naming
constants, 2–10
units, 2–9
variables, 2–10
Nested queries, 9–11
NNO variable, 9–2, C–6
default update, 9–4, 9–5
NOISE instruction, 8–5, A–7
Noise words, 8–5
NOK variable, 9–2, C–6
default update, 9–4
NOKFIRST variable, 9–2, C–7
NORMAL mode
example, 5–15
Normalized coordinates, 3–8, 3–11
NOWORD instruction, A–7
NUMBER function, 5–18, B–3
NUMTRIES variable, 9–2, 9–11, C–7
default update, 9–3
example, 9–11
passing to another unit, 9–3

O

OBJ/NOOBJ compiler switch, 6–8
Odd-even pairs, 10–10
OKWORD instruction, A–7
OLD_VERSION compiler switch, 6–8
ON instruction, 11–16, A–7
ONCHANNEL variable, C–11
ONCODE variable, C–12
ONKEY variable, C–12
OPEN instruction, 13–6, A–7
indexed files, 13–15
random files, 13–11
sequential files, 13–8
Operators
arithmetic, 2–8
Boolean, 2–8, 5–33
relational, 5–32
OTHER instruction, 5–34, A–7
OUTLOOP instruction, A–7

P

Parameter passing, 14–1
examples, 14–4
from DAL routines, 14–2
to DAL routines, 14–1
PATTERN instruction, 10–12, 10–15, A–7
example, 10–19
PAUSE instruction, 4–5, 4–6, A–7
example, 4–7
Permanent variables, 2–6
PICFILE function, B–4
Professional terminal
color capabilities, 7–5
VAX DAL on a, 7–3
Prompt character, 4–2, 5–10, 5–14, 8–14
PROMPT instruction, 5–14, 8–14, A–7
example, 8–9
PROMPT variable, C–5
PUBLISH compiler switch, 6–8
PUT instruction, 13–7, A–7
indexed files, 13–19
random files, 13–13
sequential files, 13–10

Q

QELAPSED variable, 9–6, 9–8, C–5
QLENGTH variable, 9–6, 9–7, C–5
QUERIES variable, C–7
QUERY blocks
See also response-judging
nested queries, 9–11
QUERY instruction, 4–2, A–8
example, 5–11, 8–9

R

RAD function, B–4
Rainbow terminal
  color capabilities, 7–5
Random access to records, 13–4
Random files
  access to records, 13–4
  file I/O, 13–11
  structure, 13–4
RANDOMN function, B–4
RANDOMP function, B–4
RANDOMU function, B–4
  example, 5–20
RAT instruction, 10–23, A–8
RBOX instruction, 10–23, A–8
  example, 10–30
RCIRCLE instruction, 10–23, A–8
RCURVE instruction, 10–23, A–8
RDOT instruction, 10–23, A–8
REAL function, B–4
Real numbers, 2–4
Record structures, 13–16
REDO instruction, A–8
REGIS instruction, A–8
Related documents, ii
Relative graphics, 10–23
RELOOP instruction, A–8
REPLACE function, B–4
Response, 1–1, 4–1, 4–7
  extra words, 4–7, 8–2
  modifying display of a, 7–6
  precise matching, 8–3
  punctuation in, 4–7, 8–3
  spelling of, 4–7, 8–2
  student variables, 8–9
  tolerance of machine error in, 8–3
  unit conversion in, 8–1
  units of measure, 8–3, 8–12
  uppercase and lowercase, 4–7, 8–1
  use of system functions in, 8–2
  variables used in, 8–2
  word order, 4–7, 8–1
Response judging, 1–2, 1–4
  default sequence, 9–3
  modifications, 5–21, 8–1, 8–7
  noise words, 8–5
  QUERY blocks, 4–1
Response judging (Cont.)
  specifications, 4–7, 5–9, 8–1
  specifying answers, 5–18
  synonyms, 8–4
Response processing
  error detection with ERRORV, 9–14
Response scoring
  system variable update, 9–3
RESPONSE variable, 5–18, 9–6, C–5
  default update, 9–7
Response-contingent instructions, 4–1, 4–5, 5–8, 5–18, 8–7
Response-judging block, 4–1, 4–2, 5–8
Response-related variables, 9–5
RESPONSEV variable, 5–38, 8–9, 8–13, 9–7, C–5
  default update, 9–7
Restart variables, 2–6
RESTORE instruction, 7–13, A–8
RETURN instruction, 11–3, A–8
REWIND function, 13–10, B–4
RIGHT instruction, 4–4, A–8
  example, 5–11
  noise words, 8–5
  synonyms, 8–5
RIGHTV instruction, 5–19, 5–21, 8–8, A–9
  tolerance, 8–11
  units, 8–12, 8–13
RLINE instruction, 10–23, A–9
RMSSTATUS variable, C–12
RORIGIN instruction, 10–23, 10–24, A–9
RORIGINX variable, C–2
RORIGINY variable, C–2
ROTATE instruction, 10–23, 10–28, A–9
ROUND function, B–4
Row and column addresses, 3–2, 3–3, 3–10
RSIZE instruction, 10–23, 10–26, A–9
  example, 10–27
RSIZEX variable, C–2
RSIZEY variable, C–2
Run-time errors, 6–5
RVECTOR instruction, 10–23, A–9
RWHEREX variable, C–3
RWHEREY variable, C–3

S

SATISFIED variable, 9–3, C–7
  default update, 9–3
  passing to another unit, 9–3
SAVE instruction, 7–11, A–9
Saving and restoring display attributes, 7–11
SCALEY instruction, A–9
SCORE instruction, 5–6, A–9
  example, 5–23
SCORE UPDATE instruction, 9–5, A–9
SCORE variable, 2–8, 5–22, 9–3, C–7
  default update, 9–4
SCORES variable, 5–22, 5–32, 9–3, C–8
  default update, 9–4
Scoring system variables, 9–2
Screen addresses, 3–1
  conversion, 3–11
  fine coordinates, 3–4, 3–10
    example, 10–19
  normalized coordinates, 3–8, 3–11
  row and column coordinates, 3–2, 3–10
  syntax, 3–11
SEED instruction, 5–20, A–9
Sequential access to records, 13–3
Sequential files
  access to records, 13–3
  file I/O, 13–8
  structure, 13–3
SET DELETE instruction, 7–10
SET ECHO instruction, 7–6, A–10
SET FKEY instruction, 7–8, A–10
SET HLS instruction, 7–8, A–10
SET KEYPAD instruction, 7–10, A–10
SET MAXCOLORS instruction, 7–7, 12–6, 12–11, A–10
SET TYPEAHEAD instruction, 7–6
Setting terminal characteristics, 7–1
Shading figures, 10–13
SIGN function, B–4
SIGNAL instruction, 11–20
SIN function, B–4
SIZE instruction, 3–13, 3–18, 5–13, A–10
  example, 3–12
SIZEX variable, 10–32, C–3
SIZEY variable, 10–32, C–3
SLIDE instruction, A–10
Source file, 6–1
SPEAK instruction, A–10
Special function keys, 7–8
SPECS instruction, 8–1, A–10
  example, 9–9
SPECS MACHTOL instruction, 8–11
SPECS NOCONV instruction in only some of their responses, you can, 8–12
SPECS PRECISE instruction, 8–9
SQRT function, B–4
SREF instruction, 10–13, A–11
STOP instruction, A–11
STRING function, B–4
Strings, 2–4
Student, 1–1, 2–1
Student response
  See Response
Student variables, 2–5, 8–2, 8–9
SUBSTR function, B–5
SYN instruction, 8–4, 8–5, A–11
Synonyms, 8–4, 8–5
Syntax, 2–11
  graphics instructions, 3–4, 3–7
  screen addresses, 3–11
  text block, 3–15
System constants
  mode, 3–26
System variables, 2–6
  graphics, 2–6, 10–31, 10–34, C–1
  miscellaneous, C–8
  response-related, 2–6, C–5
  saved in nested queries, 9–13
  scoring, 2–6, 9–1, C–7
  use in lessons, 9–9

T

TAN function, B–5
Terminal management, 7–1
  instructions, 7–6
Terminal models, 7–1
  color capabilities, 7–4
  DAL support requirements, 7–2
  screen size differences, 7–3
TERMSTAT variable, C–12

TEST instruction, 5–25, 5–34, A–11
  example, 9–11
Text
  block, 3–15
  size, 3–2, 5–9, 5–13
TIME variable, C–12
TIMEOUT variable, 9–7, 9–8, C–6
TRAN function, B–5
TRAY instruction, A–11
TROTATE instruction, 10–17, 10–18, A–11
  example, 10–19
TSA variable, C–12

U

Unit, 1–4, 2–1, 2–2, 5–2, 5–4
  calling chain, 11–2
  level, 5–4
  names, 2–9
UNIT instruction, 5–2, 5–4, 5–11, 5–20, A–11
Unit-level variables, 2–10, 5–26
Units of measurement, 8–12
UNITS/NOUNITS compiler switch, 6–7
UPDATE instruction, 13–7, A–11
  indexed files, 13–20
UPPER function, B–5
Uppercase, 2–11
Usage characteristics, 2–5
  EXTERNAL, 2–5
  GLOBAL, 2–5
  PERMANENT, 2–6
  RESTART, 2–6
  STUDENT, 2–5, 8–9
USERTYPE variable, C–13

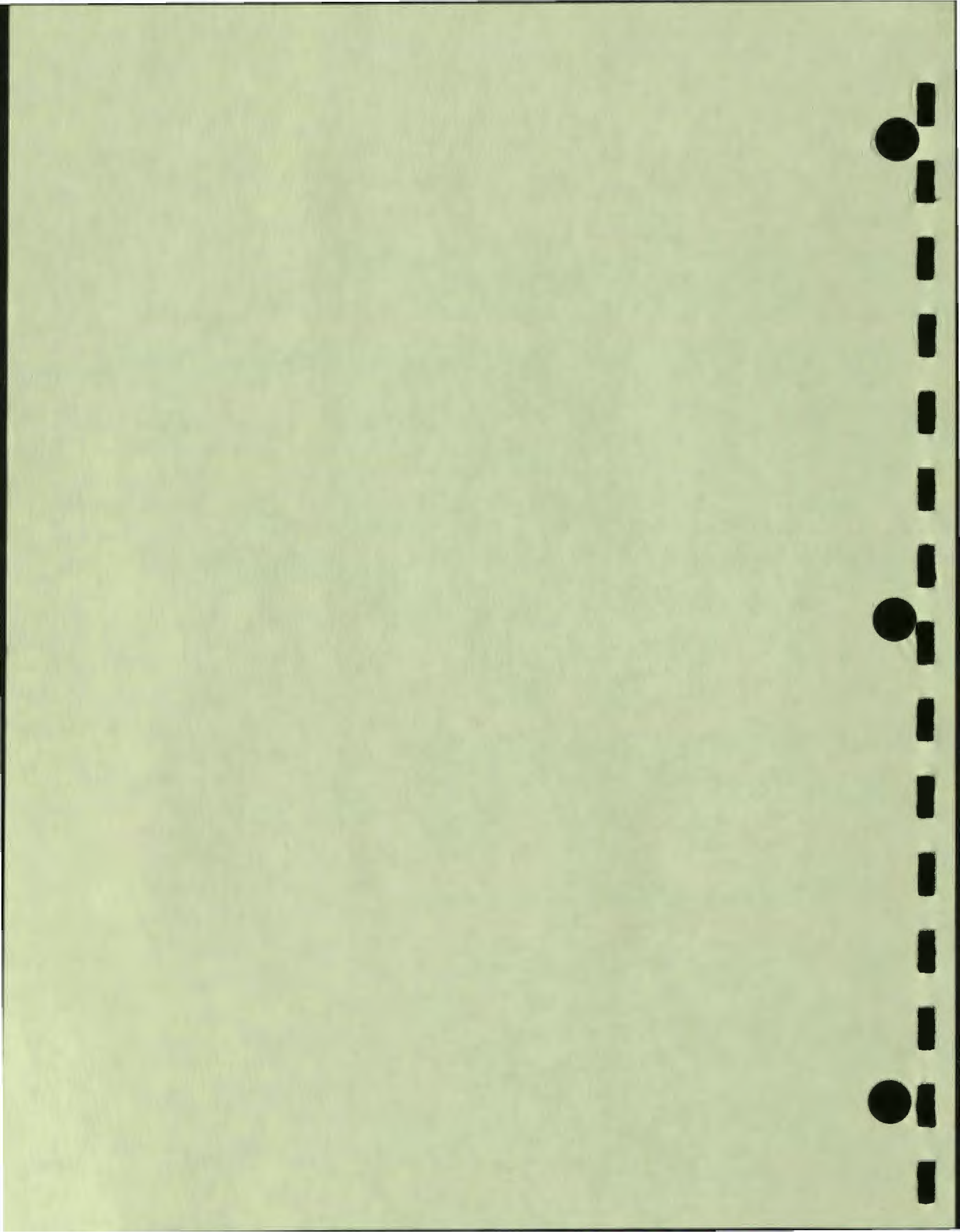
V

VALUE instruction, 5–34, A–11
  example, 9–11
Variables, 2–3, 5–28
  BOOLEAN, 5–6
  definition of, 5–6
  EXTERNAL, 2–5
  GLOBAL, 2–5
Variables (Cont.)
  INTEGER, 5–6
  lesson-level, 2–10, 5–5, 5–38
  names, 2–10
  PERMANENT, 2–6
  RESTART, 2–6
  scoring, 5–22
  STUDENT, 2–5, 8–9
  system, 2–6
  unit-level, 2–10, 5–26, 10–34
  usage characteristics, 2–5
  user-defined, 2–3, 5–18
VAX DAL color management system, 12–1
VAX DAL compiler, 6–1
  See also compiler
VAX DAL macro preprocessor, 15–1
VAX DAL run-time library, 6–9
VAX/VMS
  linker, 6–9
  run-time library routines, 14–6
  system services, 14–7
VBAR instruction, A–11
VECTOR instruction, 10–2, 10–10, A–11
  example, 3–7
VT125 terminal
  color capabilities, 7–5
  VAX DAL on a, 7–2
VT240 and VT241 terminals
  color capabilities, 7–5
  VAX DAL on, 7–2

W

WEIGHT instruction, A–12
WHEN instruction, 11–10, A–12
  disabling a, 11–11
  scope, 11–12
WHERE variable, 10–32, C–3
  example, 10–34
WHEREX variable, 10–32, C–4
WHEREY variable, 10–33, C–4
WORDS variable, C–6
WRITC instruction, 5–17, 5–28, A–12
  example, 5–32

WRITE instruction, 3–3, 5–17, A–12
  example, 3–9, 4–7
  response-contingent, 4–7, 5–13, 5–18
  text blocks, 3–14, 5–37
  variables, 3–15
    string format, 3–18, 5–17, 5–20
    tabular format, 3–18, 5–16,8–9
WRONG instruction, 4–4, A–12
  example, 5–11, 5–18
WRONGV instruction, 5–19, 8–8, A–12
  tolerance, 8–11
  units, 8–12



READER'S COMMENTS

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

Did you find errors in this manual? If so, specify by page.

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

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

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
- ☐ Non-programmer interested in computer concepts and capabilities

Name Date

Organization

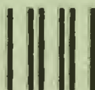
Street

City State Zip Code or Country

Please cut along this line.

Fold here

Do Not Tear – Fold Here and Staple



No Postage
Necessary
If Mailed In The
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD, MA

Postage Will Be Paid by:

digital

Software Publications
200 Forest Street MRO1–2/L12
Marlborough, Massachusetts 01752

DECLIT AA VAX K763C

VAX DAL author's guide

DECLIT AA VAX K763C

VAX DAL author's guide

SHREWSBURY LIBRARY
Digital Equipment Corporation
333 South Street SHR1-3/G18
Shrewsbury, MA 01545
(DTN) 237-3271

SHREWSBURY LIBRARY
DIGITAL EQUIPMENT CORPORATION
SHR1 3/G18
DTN 237-3400

AA-[illegible]C-TE
Printed in U.S.A.